ALIVE-INFER: Data-Driven Precondition Inference for Peephole Optimizations in LLVM

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

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

Abstract

Peephole optimizations are a common source of compiler bugs. Compiler developers typically transform an incorrect peephole optimization into a valid one by strengthening the precondition. This process is challenging and tedious. This paper proposes ALIVE-INFER, a data-driven approach that infers preconditions for peephole optimizations expressed in Alive. ALIVE-INFER generates positive and negative examples for an optimization, enumerates predicates on-demand, and learns a set of predicates that separate the positive and negative examples. ALIVE-INFER repeats this process until it finds a precondition that ensures the validity of the optimization. ALIVE-INFER reports both a weakest precondition and a set of succinct partial preconditions to the developer. Our prototype generates preconditions that are weaker than LLVM's preconditions for 73 optimizations in the Alive suite. We also demonstrate the applicability of this technique to generalize 54 optimization patterns generated by Souper, an LLVM IR–based superoptimizer.

CCS Concepts • **Software and its engineering** → **Software verification**; **Compilers**

Keywords Compilers, Alive, Learning, Inference

1. Introduction

LLVM is a widely used compiler, both in industry and academia. To attain the best possible performance, LLVM performs a large number of semantics-preserving optimizations. Among these are peephole optimizations, which perform local rewriting of code with a primary focus on algebraic simplifications. They also clean up and canonicalize code, which can enable other optimizations. In LLVM, peephole optimizations find code fragments in an input program that match a pattern, and replace them with an equivalent set of instructions. They are also a persistent source of LLVM bugs [25, 30, 51].

We have addressed the problem of peephole optimization bugs with Alive, a domain-specific language for specifying and verifying peephole optimizations in LLVM [30]. The Alive language is similar to the LLVM intermediate representation (IR). An Alive optimization has a source pattern and a target pattern, with an optional precondition (see Section 2). The Alive interpreter checks the correctness of an optimization using satisfiability modulo theories (SMT) solvers. Alive has discovered numerous bugs and is currently used by LLVM developers [18, 28, 30, 42].

Alive prevents the inclusion of wrong optimizations in the LLVM compiler. It also provides counterexamples for wrong optimizations. The developer must exclude all inputs that make the optimization invalid. Developers typically accomplish this by strengthening the precondition of the optimization. However, developing an appropriate precondition when presented with a Alive counterexample can be tedious. To illustrate, let us consider the following peephole optimization (presented in Alive syntax), which was submitted as a code patch for the LLVM compiler [18]:

```
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
```

The patch was rejected because Alive found it to be invalid and provided a counterexample. After multiple revisions, the developer found a precondition that made the optimization valid:

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

PLDI'17, June 18–23, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062372

The precondition of an optimization is a collection of predicates involving symbolic constants, constant expressions, and constant functions (see Figure 1). It determines when the optimization can be applied to an input program. A strong precondition prevents the application of the optimization for some programs where it would be valid. For example, the precondition in the LLVM patch above rules out many valid valuations for the symbolic constants. A weaker precondition that has too many predicates (not succinct) can increase compilation time, because the precondition has to be evaluated for every potential application site. The optimization developer considers trade-offs between the strength of the precondition and its succinctness. Hence, identifying preconditions for these optimizations is challenging.

This paper proposes ALIVE-INFER, a data-driven approach for identifying appropriate preconditions for LLVM peephole optimizations expressed in Alive. ALIVE-INFER is inspired by PIE (Precondition Inference Engine) [39] and other data-driven approaches [13–15, 43, 47] to generate preconditions and loop invariants for general purpose programs. We design new techniques and adapt the PIE approach to address the following challenges in the Alive context. First, we must address compile-time undefined behavior in the constant expression language. We want to reason about potentially unsafe predicates without the risk of crashing LLVM at compile-time. Second, we must generate examples (data) in a static setting to use a data-driven approach. Third, we must handle type polymorphism in Alive while generating examples and enumerating predicates. Finally, we must address the trade-off between succinct and widely-applicable preconditions.

ALIVE-INFER addresses these challenges and proposes an end-to-end solution for generating preconditions that developers can use. We divide the task of inferring preconditions for an optimization into three subtasks: (1) example generation, (2) predicate enumeration and learning, and (3) Boolean formula learning.

Example generation. ALIVE-INFER's example generator creates positive and negative examples for an optimization. We propose that an example in this setting should provide types and valuations for the symbolic constants. ALIVE-INFER must consider types while generating examples because Alive optimizations are parametric types. It must ensure that no positive example causes compile-time undefined behavior. ALIVE-INFER generates examples through random selection and by querying an SMT solver. It classifies an example as positive if the refinement check for the optimization is valid on substituting the symbolic constants with concrete values, which is checked using an SMT solver. The refinement check has a for-all quantification for the input variables. To ensure sufficient number of positive and negative examples, ALIVE-INFER supplements the randomly-chosen examples with examples obtained using an SMT solver (see Section 3.2).

Predicate learning. Next, ALIVE-INFER enumerates and learns a set of predicates to accept all positive examples and reject all negative examples. As with PIE [39], new predicates are learned on-demand by enumerating predicates and evaluating them on a small sample of examples. The enumerator lazily produces polymorphic, type-correct predicates. ALIVE-INFER must consider whether a predicate can be evaluated safely. For a given example, a predicate may be true, false, or unsafe. ALIVE-INFER learns predicates which separate the positive and negative examples in the sample. These narrow the search for future predicates, and may be used in the precondition (see Section 3.3).

Boolean formula learning. Once ALIVE-INFER learns sufficiently many predicates, it uses one of two Boolean formula learners to assemble a precondition. The full Boolean learner produces preconditions which accept all positive examples and reject all negative examples, but may be large and complex. In contrast, the partial Boolean learner produces succinct preconditions which may not accept all positive examples. ALIVE-INFER reports multiple partial preconditions to the developer as they are obtained, and terminates once it produces a full precondition that is proven to accept all positive examples, or *weakest*. This provides developers a choice between applicability and succinctness. The Boolean learner must ensure compile-time safety of the learned precondition. The presence of unsafe predicates introduces new challenges in formula learning as the negation of an unsafe predicate is still unsafe (see Section 3.4). A partial precondition accepts a subset of the positive examples while rejecting all negative examples. The Boolean learner attempts to maximize the number of positive examples accepted while generating partial preconditions. In contrast, a weakest precondition accepts all positive examples and rejects all negative examples. ALIVE-INFER checks the validity of both partial and weakest preconditions, and whether a proposed weakest precondition rejects any positive examples.

We built the ALIVE-INFER prototype for generating preconditions by extending the publicly-available Alive-NJ toolkit [33]. We evaluated it using the Alive suite of optimizations. Out of the 417 optimizations in the Alive suite, there are 174 optimizations that have a precondition. The ALIVE-INFER prototype generates the weakest precondition for 133 of them within 1000 seconds. It generates either a partial or the weakest precondition for 164 out of the 174 optimizations. ALIVE-INFER generates a weaker precondition than the precondition in the Alive suite for 73 optimizations. We have also used ALIVE-INFER to generalize concrete optimization patterns generated by Souper [21], an LLVM IR–based superoptimizer (see Section 4). We generalized a total of 71 optimizations that are expressible in Alive. ALIVE-INFER is able to generate preconditions for 54 of them.

2. Background on Alive

Alive [30] is a domain-specific language for specifying and verifying peephole optimizations in LLVM. The Alive interpreter checks the correctness of an Alive optimization by encoding it as constraints, which allows automated reasoning with SMT solvers. The interpreter also generates C++ code when the optimization is correct. To encourage adoption by LLVM developers, the Alive language is similar to the LLVM intermediate representation. Alive has found numerous bugs in the LLVM compiler [18, 30]. LLVM developers are actively using Alive to check the correctness of new optimizations (patches) submitted to LLVM. Alive based tools have prevented many bugs in patches committed to LLVM [18, 28, 42]. Although C++ code generation is not actively used, there are plans for replacing InstCombine with Alive-generated C++ code. Alive has also been extended to reason about the correctness of floating point optimizations (*e.g.*, Alive-FP [36] and LifeJacket [38]). We describe the language and the verification process next.

The Alive language. An Alive optimization has the form `source` $\Rightarrow$ `target`, with an optional precondition. The source and target describe directed, acyclic graphs (DAGs) of values. Semantically, an Alive optimization replaces the DAG in the source with the DAG in the target. The interior nodes correspond to Alive (LLVM IR) instructions, with incoming edges representing their arguments. Leaf nodes are input variables. They represent arbitrary LLVM values, such as results from other instructions, symbolic constants, constant expressions, and function parameters. We will write $\mathcal{C}$ for the set of input variables whose values are known while compiling a concrete input program. Types for the variables, values for symbolic constants and constant expressions are available during compilation. We use $\mathcal{R}$ to represent remaining input variables that are not known at compile time.

An example Alive optimization is shown in Figure 4(a) and its DAG representation is shown in Figure 4(b). There are three input variables: `%X`, `C1`, and `C2`, where `%X` is a run-time input variable, `C1` and `C2` are symbolic constants, and `C1 /u C2` is a constant expression.

Preconditions. A precondition for a peephole optimization in LLVM is checked during compilation of an input program before applying the optimization. Hence, preconditions for these optimizations primarily deal with values that can be determined during compilation: types, symbolic constants, and constant expressions. Figure 1 provides the abstract syntax of preconditions for LLVM peephole optimizations. A precondition is a conjunction or disjunction of various predicates. A predicate is either a predicate function or a binary comparison operation involving constant expressions. Constant expressions can be symbolic constants, constant functions, and binary operations of constant expressions.

Verification of an optimization. As Alive optimizations are polymorphic over types, the Alive interpreter checks the correctness of the optimization for each feasible type (up to a bound on integer width). Alive models various kinds of undefined behavior in LLVM (*i.e.*, poison values, undef values, and true undefined behavior) [30]. The subtleties of the semantics are currently being explored [29]. For simplicity, we use a definedness constraint for an instruction to exclude all kinds of undefined behavior while describing the verification below.

$$
\begin{array}{rcl}
\mathit{pre} & ::= & \mathit{pred} \mid \neg \mathit{pre} \mid \mathit{pre} \wedge \mathit{pre} \mid \mathit{pre} \vee \mathit{pre} \\
\mathit{pred} & ::= & \mathit{binpred} \mid \mathit{pfun} \\
\mathit{binpred} & ::= & \mathit{cexpr}\ \mathit{cond}\ \mathit{cexpr} \\
\mathit{cexpr} & ::= & \mathit{constant} \mid \mathit{unop}\ \mathit{cexpr} \mid \\
 & & \mathit{cexpr}\ \mathit{binop}\ \mathit{cexpr} \mid \mathit{cfun} \\
\mathit{cond} & ::= & \textbf{eq} \mid \textbf{ne} \mid \textbf{ugt} \mid \textbf{uge} \mid \textbf{ult} \mid \\
 & & \textbf{ule} \mid \textbf{sgt} \mid \textbf{sge} \mid \textbf{slt} \mid \textbf{sle} \\
\mathit{binop} & ::= & \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{udiv} \mid \textbf{sdiv} \mid \\
 & & \textbf{urem} \mid \textbf{srem} \mid \textbf{shl} \mid \textbf{lshr} \mid \textbf{ashr} \mid \\
 & & \textbf{and} \mid \textbf{or} \mid \textbf{xor} \\
\mathit{unop} & ::= & \textbf{neg} \mid \textbf{not} \\
\mathit{cfun} & ::= & \textbf{abs}\ \mathit{cexpr} \mid \textbf{log2}\ \mathit{cexpr} \mid \textbf{width}\ \mathit{value} \\
\mathit{pfun} & ::= & \textbf{isSignBit}\ \mathit{cexpr} \mid \textbf{isPowerOf2}\ \mathit{cexpr} \mid \\
 & & \textbf{isPowerOf2OrZero}\ \mathit{cexpr}
\end{array}
$$

Figure 1. Abstract syntax of preconditions.

For each feasible type assignment, Alive creates two expressions for each instruction: ι, the value it returns, and δ, the necessary conditions for it to have well-defined behavior. The interpreter also generates an SMT expression ϕ corresponding to the precondition. A transformation is correct if and only if the target is defined and the roots of the source and target DAGs produce the same value when the precondition is satisfied and the source is defined. That is:

$$\forall_{\mathcal{R},\mathcal{C}} : \phi \wedge \delta_s \implies \delta_t \wedge \iota_s = \iota_t,$$

where $\mathcal{R}, \mathcal{C}$ is the set of input variables in the DAG, δ_s and δ_t are constraints for the source and the target to be have defined behavior, respectively, and ι_s and ι_t are the values computed by the source and target.

3. Precondition Inference

ALIVE-INFER is an end-to-end solution that infers preconditions for LLVM optimizations expressed in Alive. Our approach is inspired by PIE [39], a data-driven approach for inferring preconditions and loop invariants for general-purpose programs. However, we design new techniques and algorithms to address the following challenges in the Alive context: addressing compile-time undefined behavior in Alive, generating data in a static setting, handling type polymorphism, and generating succinct partial preconditions.

ALIVE-INFER consists of three components: (1) the example generator, which produces a set of positive and negative examples, (2) the predicate learner, which learns a set of predicates that can separate the positive and negative examples,

```
function InferPrecondition(opt, I)
    ⟨E+, E−⟩ ← MakeExamples(opt)
    P_valid ← ∅
    repeat
        ⟨P_p, P_f⟩ ← PreconditionsByExamples(E+, E−, I)
        e− ← ∅
        for all p ∈ P_p do
            e_p− ← CounterExamples(p, opt)
            if e_p− = ∅ then
                P_valid ← P_valid ∪ {p}
            e− ← e− ∪ e_p−
        e_f− ← CounterExamples(P_f, opt)
        e− ← e− ∪ e_f−
        e+ ← ∅
        if e_f− = ∅ then
            P_valid ← P_valid ∪ P_f
            e+ ← PositiveExamples(P_f, opt)
        E+ ← E+ ∪ e+
        E− ← E− ∪ e−
    until e− = e+ = ∅
    return P_valid
```

Figure 2. Algorithm for generating preconditions for an LLVM peephole optimization *opt* with an initial set of predicates I. We generate an initial set of examples with the function MAKEEXAMPLES. The function PRECONDITIONSBYEXAMPLES enumerates predicates on-demand and returns a tuple: (a set of partial preconditions and a complete precondition for the sample). Both the partial preconditions and the complete precondition are checked for validity and counter examples are added to the set of bad examples. If the complete precondition is valid, it checks if it is the weakest.

and (3) the Boolean formula learner, which learns a Boolean formula in conjunctive normal form (CNF).

Figure 2 provides a high level sketch of our approach. ALIVE-INFER's example generator addresses the challenge of generating data in a static setting where input programs are not available. It ensures that the feasible types of an optimization are sufficiently represented in the set of examples. It also ensures that examples do not cause any compile-time undefined behavior. The developer can also guide the example generation process. ALIVE-INFER's predicate learner synthesizes new predicates while accounting for types in Alive and learns a set of predicates that are useful in separating the positive and the negative examples. It must account for safety of the predicate at compile time. ALIVE-INFER's Boolean formula learner generates either a partial or weakest precondition using the learned predicates. The Boolean learning algorithms need to account for the safety of the learned preconditions. ALIVE-INFER checks the validity of the optimization with both partial and weakest preconditions. If the optimization is not valid with the learned precondition, ALIVE-INFER adds counterexamples to the set of negative examples and repeats the process. When the optimization is valid but there are positive examples that are disallowed by the precondition, ALIVE-INFER reports the partial precondition to the developer and adds the positive examples to the set of positive examples and repeats the process to weaken the precondition. Next, we describe compile-time undefined behavior, which influences all components of ALIVE-INFER.

3.1 Addressing Compile-time Undefined Behavior

Alive's constant expression language includes operations that are not defined for all possible inputs. The semantics for Alive's integer expressions are based on the corresponding semantics for SMT bitvector operations and LLVM's arbitrary-precision integer library. Both include operations that are undefined in certain circumstances, such as division by zero. Verifying an optimization which is not fully defined may result in unexpected or inconsistent solver behavior. Performing such an optimization in LLVM using Alive-generated code may result in a compiler crash.

For example, the precondition `C1 % C2 == 0` is undefined when `C2` is zero. We say it has *compile-time undefined behavior*, to distinguish it from the undefined behavior present in LLVM IR. In particular, Alive-generated code in LLVM may crash when evaluating the precondition.

We associate a *safety condition* with each Alive term. This condition is true if and only if the term does not have compile-time undefined behavior. For our example, the safety condition is $\texttt{C2} \neq 0$. Calculation of safety conditions is mostly straightforward, but there are some subtleties. The precondition `C2 != 0 && C1 % C2 == 0` is always safe, because the corresponding SMT expression is well-defined and the C++ translation will avoid dividing by zero due to short-circuit evaluation.

We extend the refinement condition for Alive to include safety conditions. The source of an optimization contains no constant expressions, so it is trivially safe. The precondition must always be safe to evaluate. Any constant expressions in the target must be safe when the precondition is satisfied. The new refinement condition for checking the correctness of an Alive optimization is:

$$\forall_{\mathcal{R},\mathcal{C}}\ \sigma_\phi \wedge (\phi \implies \sigma_t \wedge (\delta_s \implies \delta_t \wedge \iota_s = \iota_t)), \quad (1)$$

where ϕ, ι_s, ι_t, δ_s, δ_t, σ_t, and σ_ϕ are constraints to represent the precondition, the value produced by source, value produced by the target, definedness constraints for the source, definedness conditions for the target, safety conditions for the compile time constant expressions in the target, and safety conditions for the constant expressions in the precondition, respectively. Here, $\mathcal{R}$ and $\mathcal{C}$ represent the set of input variables and symbolic constants, respectively. We have changed Alive-NJ to use the new refinement condition for verification. Each component of ALIVE-INFER has to consider safety while learning a precondition.

3.2 Example Generation

One contribution of ALIVE-INFER is the use of a data-driven approach to infer preconditions in the context of compiler

verification, where concrete input programs are not available. ALIVE-INFER has to generate examples in this setting. The key challenges in example generation are: handling type polymorphism in Alive, identifying a method to classify an example as positive or negative, and methods to quickly generate sufficient and diverse examples.

What is an example? An example in our setting represents an input program that matches the source of an optimization. The precondition determines when an optimization can be performed using the information available while compiling a concrete input program: the types of the source values and the values of the symbolic constants. Hence, examples in ALIVE-INFER contain type assignments and values for the symbolic constants (consistent with their assigned types).

ALIVE-INFER uses examples with different type assignments to avoid creating preconditions that are only valid for one assignment. For example, some operations may overflow at small types, but not at large types. Additionally, having examples at different types increases the chances of learning predicates that vary based on types, such as `width(%a)`.

Positive and negative examples. We classify an example as *positive* if the optimization's target refines the source and has no compile-time undefined behavior when the symbolic constants in the optimization are substituted with concrete values from the example. Otherwise, it is *negative*. Any example that causes compile-time undefined behavior or fails the refinement check is a negative example, which should be disallowed by the learned precondition. Given sets of runtime variables $\mathcal{R}$ and symbolic constants $\mathcal{C}$, we simplify the refinement check in Section 3.1 and define

$$V(\mathcal{C}, \mathcal{R}) \equiv \sigma_t \wedge (\delta_s \implies \delta_t \wedge \iota_s = \iota_t). \tag{2}$$

We use $V(e, \mathcal{R})$ to represent the substitution of symbolic constants with concrete values from the example e in the above equation. Hence, an example e is positive if and only if $\forall_{\mathcal{R}} V(e, \mathcal{R})$.

Methods to generate examples. To handle Alive's type polymorphism, we first sample the set of type assignments for the variables in the optimization. For a given type assignment, ALIVE-INFER obtains examples using three methods: using an SMT solver, classifying randomly-generated examples, and classifying a small set of examples using boundary values. We can obtain negative examples by passing the following negated refinement condition to the SMT solver and extracting values for the symbolic constants from the models it returns:

$$\exists_e \exists_{\mathcal{R}} \neg V(e, \mathcal{R}). \tag{3}$$

Using an SMT solver to find positive examples is similar, but we additionally require positive examples to have a well-defined source for at least one assignment of run-time variables. This is not necessary for correctness, but allowing the precondition to reject trivial positive examples can result in simpler preconditions. Thus, we obtain positive examples by using values for the symbolic constants from the models of the following formula:

$$\exists_e (\exists_{\mathcal{R}} \delta_s) \wedge (\forall_{\mathcal{R}} V(e, \mathcal{R})). \tag{4}$$

The final two methods involve first generating examples and then classifying them. We create examples by randomly choosing values for each symbolic constant in $\mathcal{C}$ that fall within its type. We create additional examples by taking the Cartesian product of $\{0, 1, -1, m\}$ for each variable in $\mathcal{C}$ (m is the minimum signed value for that type). Once we obtain a proposed example e, we use an SMT solver to check whether $\exists_{\mathcal{R}} \delta_s[\mathcal{C}/e]$, where we specialize δ_s by substituting the variables from $\mathcal{C}$ with their values from e. If not, then e is trivial and gets discarded. If so, we check whether $\exists_{\mathcal{R}} \neg V(e, \mathcal{R})$. If so, e is negative. Otherwise, e is positive.

The example generation methods have complementary benefits. Random generation of values for symbolic constants is fast but may not generate enough positive and/or negative examples. In contrast, examples generated using SMT solvers can be slow.

We cannot use a fixed number of examples for every optimization because number of type assignments are exponential in the number of feasible types for an optimization. We increase the number of examples logarithmically with the number of possible type assignments.

Developer support to guide example generation. ALIVE-INFER will eventually find a precondition which accepts all positive examples and rejects all negative examples, but sometimes the developer may wish to exclude examples from consideration. Examples may be excluded because the structure in LLVM ensures that the optimization will never be applied to them, so the precondition need not explicitly reject them. Conversely, some positive examples may represent uninteresting cases, and the precondition need not explicitly accept them. The developer can inform ALIVE-INFER about such examples using assumptions, which are terms in the precondition language. ALIVE-INFER discards any example that does not satisfy the assumptions.

3.3 On-demand Predicate Enumeration and Learning

Inspired by PIE [39], ALIVE-INFER separates predicate learning from Boolean formula learning. In contrast to PIE, ALIVE-INFER addresses the following challenges: handling predicates that are unsafe with respect to an example and enumerating type-polymorphic predicates consistent with an optimization's type constraints. Given a set of examples, ALIVE-INFER creates a sample of examples that are not currently separated by the learned predicates. It enumerates predicates on-demand until it finds a predicate that separates the positive and negative examples in the sample, meaning it accepts all the positives and rejects all the negatives, or vice versa. When it finds such a predicate, it tests the predicate

```
function PRECONDITIONSBYEXAMPLES(E+, E−, I)
    P ← I
    M ← EMPTYPREDICATEMATRIX
    for all p ∈ I do
        M ← ADDPREDICATE(p, M)
    Φ ← ∅
    while MIXEDVECTORS(M) ≠ ∅ do
        V+w, V− ← WEIGHTEDPARTITION(M)
        φ ← LEARNPARTIALBOOLEAN(P, V+w, V−, 1)
        Φ ← Φ ∪ {φ}

        Select v ∈ MIXEDVECTORS(M)
        e+, e− ← SAMPLE(v, M)
        p ← LEARNPREDICATE(e+, e−)
        P ← P ∪ {p}
        M ← ADDPREDICATE(p, M)
    V+, V− ← PARTITION(M)
    φf ← LEARNCOMPLETEBOOLEAN(P, V+, V−)
    return ⟨Φ, φf⟩
```

Figure 3. Algorithm for learning preconditions given a set of examples and an intial set of predicates (I). Function ADDPREDICATE adds a predicate to the predicate matrix. Function WEIGHTEDPARTITION partitions the predicate vectors into positive vectors and negative vectors and the weight of the positive vector is the number of positive examples accepted by the positive vector. Function LEARNPARTIALBOOLEAN computes the partial precondition using the weighted positive vectors and negative vectors (see Figure 7). When the predicate matrix does not have any mixed vectors, the weakest precondition is computed by the function LEARNCOMPLETEBOOLEAN (see Figure 5). The algorithm returns a tuple — a set of valid partial preconditions and the weakest precondition — for the given set of examples.

on the entire set of examples. When it has accumulated a set of predicates that accept some (or all) positive examples and reject all negative examples from the set of examples, it learns a Boolean formula for the precondition. Figure 3 provides a sketch of our algorithm for predicate learning.

Constructing the predicate matrix. To identify whether the algorithm has learned a sufficient number of predicates to accept all positive examples and reject all negative examples, ALIVE-INFER conceptually constructs a *predicate matrix*. The rows in this matrix correspond to the examples and the columns correspond to the currently learned predicates. The matrix is updated whenever a new predicate is learned. Figures 4(e) and 4(f) illustrate predicate matrices with one and two predicates, respectively. Each entry in the matrix is accept ($\top$), reject ($\bot$), or unsafe ($\star$). These correspond to the results of evaluating the predicate after substituting the type parameters and symbolic constants in the example: it may evaluate to true, evaluate to false, or have compile-time undefined behavior, respectively.

In Figure 4(e), the predicate `C1 u< C2` accepts the negative example `(0, 1)` and rejects the negative example `(4, 2)`. Here, `u<` is the unsigned comparison operation. In Figure 4(f), the predicate `C2 /u C1 == 0` is unsafe with the example `(0, 1)` because it causes compile-time undefined behavior (division by zero).

Predicate vectors. Each row in the predicate matrix contains the results of applying the learned predicates to a particular example. We call such a list of results a *predicate vector*. The same vector may be associated with multiple examples. We call a vector that is only associated with positive examples a positive vector. Similarly, vectors that are only associated with negative examples are negative vectors. Vectors that are associated with both positive and negative examples are *mixed vectors*. An initial predicate matrix before any predicates have been learned associates every example with the empty vector $\langle\rangle$, which is mixed.

Generating preconditions. If the predicate matrix contains at least one positive vector, the algorithm can generate a partial precondition. This precondition will reject all negative examples, and accept some positive examples. ALIVE-INFER uses the partial Boolean learner described in Section 3.4 to find a formula that accepts some positive vectors and rejects all negative and mixed vectors. Each positive vector is weighted by the number of associated examples.

If the predicate matrix contains no mixed vectors, the algorithm can generate a precondition using the complete Boolean learner. This precondition will reject all negative examples, and accept *all* positive examples. Once a complete precondition is found, ALIVE-INFER checks if it is valid and weakest. If not, it adds the new examples and continues.

Otherwise, the predicate matrix contains at least one mixed vector, so ALIVE-INFER must learn more predicates.

Learning predicates. The purpose of introducing new predicates is to reduce the number of examples associated with mixed vectors—ideally to zero. To do this, ALIVE-INFER selects a mixed vector and searches for a predicate which separates its associated positive and negative examples, meaning that the predicate accepts the positive examples and rejects the negative examples, or vice versa. If the vector has many examples, it may be difficult to find such a predicate, so ALIVE-INFER searches for one which separates "enough" examples. This is particularly important early on, when all examples are associated with the empty vector. Without sampling, ALIVE-INFER would have to find a single predicate which expresses the entire precondition.

Given a mixed vector, ALIVE-INFER selects a certain number of examples associated with the vector. These examples are called the *sample*. The sample always contains both positive and negative examples. If the mixed vector has only a few examples, then all are included in the sample.

Once the sample is selected, ALIVE-INFER enumerates predicates until it finds one which separates the sample. We permit the predicate to be unsafe for negative examples in the sample, but forbid the predicate to be unsafe for any positive example. This simplifies precondition generation later on, as

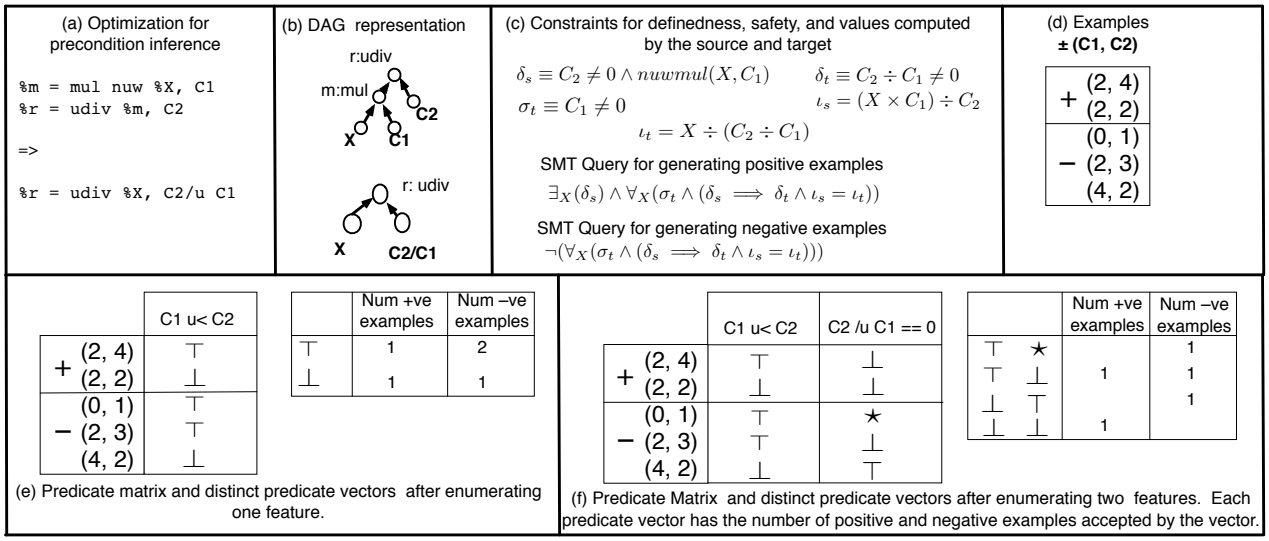


Figure 4. The process of learning preconditions. (a) LLVM peephole optimization expressed in Alive whose precondition is being learned. (b) DAG representation of the optimization, which has input runtime variable X and symbolic constants `C1` and `C2`. (c) Constraints for the definedness of the source (δ_s), definedness of the target (δ_t), compile-time safety of the target (σ_t), value produced by the source (ι_s), and value produced by the target (ι_t). Queries provided to SMT solvers to generate positive and negative examples are also provided. The predicate $nuwmul(X, C_1)$ encodes the fact that X multiplied by C_1 can be represented as an unsigned integer in the current type assignment. (d) Sample set of examples generated. We omit type assignments for simplicity. An example `(4, 2)` represents a positive example with $C_1 = 4$ and $C_2 = 2$. Any example with $C_2 = 0$ will be discarded, as it causes undefined behavior in the source. In contrast, any example with $C_1 = 0, C_2 \neq 0$ will be marked negative, because it causes an unsafe computation in the target. (e) The predicate matrix and distinct predicate vectors after adding the predicate `C1 u< C2`. $\top$ indicates that the predicate accepts the example. $\bot$ indicates that the predicate rejects the example. $\star$ indicate that the example is unsafe (compile-time undefined behavior). (f) Predicate matrix after adding two features. Our incomplete boolean learner will find a subset of the predicates `C1 u< C2`, `C1 u>= C2`, `C2 /u C1 == 0`, and `C1 /u C2 != 0`, which accepts as many of the positive vectors as it can and rejects all the negative and mixed vectors. For this matrix, it will produce the precondtion `(C1 u>= C2) && (C2 /u C1 != 0)`.

ALIVE-INFER is free to assume that unsafe examples can be rejected. If the predicate meets these conditions, it is added to the list of learned predicates and the predicate matrix is extended by evaluating the predicate on all examples. At this point, ALIVE-INFER checks to see whether it can generate new preconditions, as described earlier.

Type-aware predicate enumeration. ALIVE-INFER generates predicates using bounded recursion. Each predicate is assigned a weight, and ALIVE-INFER enumerate predicates with increasing weights. The weight of a predicate roughly corresponds to the number of leaf nodes in the abstract syntax tree (AST) of the predicate in Alive's internal representation. Exceptions are predicate functions and constant functions, which contribute to the weight but are not leaf nodes.

The enumerator is aware of the type constraints that should be satisfied by predicates and expressions. For example, the arguments to a comparison must have the same type. Enumeration of constant expressions is type-directed: the enumerator takes the desired type expression as a parameter, it propagates this type into subexpressions where necessary, and it selects appropriately-typed symbols when it reaches a leaf node.

function LEARNCOMPLETEBOOLEAN($Preds, V^+, V^-$)
  $lits \leftarrow Preds \cup \{\neg p : p \in Preds\}$
  $k \leftarrow 0$
  $C \leftarrow \emptyset$
  while $\exists v \in V^-$ s.t. ACCEPTS($\bigwedge C, v$) **do**
    $k \leftarrow k + 1$
    $C_k \leftarrow \{\bigvee d : d \subseteq lits, |d| = k\}$
    $C \leftarrow C \cup \{d : d \in C_k, \forall v \in V^+(\text{ACCEPTS}(d, v))\}$
  return COVERCLAUSES(C, V^-)

Figure 5. Algorithm for learning a Boolean formula given a set of predicates, positive vectors, and negative vectors.

To avoid generating equivalent expressions (*e.g.*, $a + (b + c)$, $(a + b) + c$, $(b + a) + c$), our enumerator is aware of the algebraic properties of the predicate language, and produces expressions in a normal form. However, we have to be careful in applying only algebraic identities with bitvector arithmetic. For example, $-(a \div b)$, $-a \div b$, and $a \div -b$ are all distinct expressions with bitvector arithmetic.

3.4 Boolean Formula Learning with Weighted Vectors

Once ALIVE-INFER has found a set of predicates and their behavior for each example, it assembles these predicates into a Boolean formula using conjunction, disjunction, and negation. ALIVE-INFER uses two different methods for learning Boolean formulae. Both learn formulae that reject all negative examples. One learns a possibly-large formula that accepts all positive examples. The other learns a formula that covers as many positive examples as it can while remaining succinct. Both produce formulae in conjunctive normal form (CNF).

The learners do not need to know about the specific examples used or predicates learned during inference. Instead, their inputs are the unique predicate vectors from the predicate learner's predicate matrix. Recall that a predicate vector is an ordered list describing the behavior of each predicate when evaluated on an example. Figures 4(e–f) show some examples of predicate vectors. Given a set of n-entry predicate vectors, the learners create a formula using abstract predicates $p_1, \ldots, p_n$, which will later be replaced by the corresponding learned predicates. The behavior of p_i for a vector v is determined by v_i. The learned formulae will accept one or more vectors associated only with positive examples, and reject all vectors associated with negative examples.

The simplest method for finding a Boolean formula would be to translate each positive vector into a conjunctive clause that accepts only that vector, and then take the disjunction of such clauses for all positive predicate vectors. For example, with positive vectors $\top\bot\top$ and $\top\bot\bot$ we might learn $(p_1 \wedge \neg p_2 \wedge p_3) \vee (p_1 \wedge \neg p_2 \wedge \neg p_3)$, which accepts only those vectors. While very simple, this method may produce needlessly complex formulae, cannot exclude unnecessary predicates, and does not produce formulae in CNF.

Evaluating clauses. Both learners work with clauses that are disjunctions of (possibly negated) predicates. The function ACCEPTS(c, v) determines whether a clause c accepts or rejects a vector v. It evaluates c by checking v_i for each $p_i \in c$, and rejects if all p_i reject.

In contrast to PIE, ALIVE-INFER can reason about predicates that are unsafe ($\star$), meaning they exhibit undefined behavior during evaluation instead of evaluating to accept ($\top$) or reject ($\bot$). The predicate enumerator ensures that no predicate is unsafe for any positive vector, so the Boolean learners are free to assume that any clause which exhibits unsafe behavior for a vector will reject that vector. In particular, p_i and $\neg p_i$ both reject a vector v where $v_i = \star$.

Because unsafe predicates are handled by ACCEPTS and in the predicate learner, the Boolean learners need not be aware of unsafe predicates.

Complete Boolean formula learning. ALIVE-INFER finds formulae in two stages. First, it chooses disjunctive clauses of up to k predicates that accept all positive vectors. As shown in Figure 5, it begins with $k = 1$ and iteratively increases k until every negative vector is rejected by at least one chosen clause.

function COVERCLAUSES(C, V^-)
  $P \leftarrow \top$
  while $\exists v \in V^-$ s.t. ACCEPTS(P, v) **do**
    $c \leftarrow \text{argmax}_{d \in C} |\{v : v \in V^-, \neg\text{ACCEPTS}(d, v)\}|$
    $V^- \leftarrow V^- \setminus \{v : v \in V^-, \neg\text{ACCEPTS}(c, v)\}$
    $C \leftarrow C \setminus \{c\}$
    $P \leftarrow P \wedge c$
  return P

Figure 6. Greedy set-cover algorithm that returns a set of clauses rejecting all negative examples.

function LEARNPARTIALBOOLEAN$(Preds, V_w^+, V^-, K)$
  $lits \leftarrow Preds \cup \{\neg p : p \in Preds\}$
  $D \leftarrow \{\bigvee d : d \subseteq lits, |d| \leq K\}$
  $C \leftarrow \emptyset$
  while $\exists v \in V^-$ s.t. ACCEPTS$(\bigwedge C, v)$ **do**
    $A \leftarrow \{\langle w, v\rangle : \langle w, v\rangle \in V_w^+, \text{ACCEPTS}(\bigwedge C, v)\}$
    $c \leftarrow \text{argmax}_{d \in D} \sum\{w : \langle w, v\rangle \in A, \text{ACCEPTS}(d, v)\}$
    $C \leftarrow C \cup \{c\}$
    $D \leftarrow D \setminus \{c\}$
    if $D = \emptyset$ **then**
      return $\bot$
  return COVERCLAUSES(C, V^-)

Figure 7. Algorithm for learning a partial Boolean formula that rejects all negative vectors and maximizes the weights of the positive vectors accepted.

That is, no negative vector is accepted by the conjunction $\bigwedge C$ of all chosen clauses in C. Figures 8(d–e) illustrate the process of increasing k until all negative vectors are rejected, and give the learned formula.

In the next stage, ALIVE-INFER finds a subset of C that still rejects all negative vectors. We use a greedy approximate set-cover algorithm, shown in Figure 6, which repeatedly selects the clause in C that rejects the most negative vectors that have not already been rejected until all negative vectors have been rejected.

Weighted partial Boolean formula learning. While it is always possible to find a complete Boolean formula that accepts all positive examples and rejects all negative examples, such a formula may be very complex. This is not always desirable, so ALIVE-INFER optionally reports a set of less-complex partial preconditions, which reject all negative examples but accept only some positive examples.

Our algorithm for finding partial Boolean formulas, shown in Figure 7, operates similarly to the complete Boolean learner, but limits complexity by only generating disjunctive clauses of up to K predicates, where K is a parameter. It first creates a set D of all clauses up to size K, and chooses a set $C \subseteq D$ containing clauses that accept all positive vectors. If $\bigwedge C$ is insufficient to reject all negative vectors, it chooses new clauses in D to add to C. Any new clause will reject some positive vectors. To guide the choice, ALIVE-INFER associates a weight with each positive vector, and

V^+	w	p_1	$\neg p_1$	p_2	$\neg p_2$	p_3	$\neg p_3$
$\bot\top\bot$	8		+	+			+
$\bot\top\top$	8		+	+		+	
$\top\bot\bot$	10	+			+		+
$\top\bot\top$	1	+			+	+	
$\top\top\top$	3	+		+		+	
		14	16	19	11	12	18
V^-							
$\bot\bot\bot$		−		−		−	
$\bot\bot\top$		−		−			−
$\top\top\bot$			−		−	−	

(a) Select p_2, discard $\top\bot\bot$, $\top\bot\top$, $\bot\bot\bot$ and $\bot\bot\top$.

V^+	w	p_1	$\neg p_1$	p_2	$\neg p_2$	p_3	$\neg p_3$
$\bot\top\bot$	8		+	+			+
$\bot\top\top$	8		+	+		+	
$\top\top\top$	3	+		+		+	
		3	16		0	11	8
V^-							
$\top\top\bot$			−		−	−	

(b) Select $\neg p_1$, discard $\top\top\top$ and $\top\top\bot$.

V^+	w	p_1	$\neg p_1$	p_2	$\neg p_2$	p_3	$\neg p_3$
$\bot\top\bot$	8		+	+			+
$\bot\top\top$	8		+	+		+	
		0			0	8	8
V^-							

(c) Final: $p_2 \wedge \neg p_1$.

V^-	$p_1 \vee p_2$
$\bot\bot\bot$	−
$\bot\bot\top$	−
$\top\top\bot$	

(d) 2-CNF terms

V^-	$p_1 \vee p_2$	$p_1 \vee p_2 \vee p_3$	$p_1 \vee p_2 \vee \neg p_3$	$\neg p_1 \vee \neg p_2 \vee p_3$
$\bot\bot\bot$	−	−		
$\bot\bot\top$	−		−	
$\top\top\bot$				−

(e) 3-CNF terms. Cover is $(p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_2 \vee p_3)$.

Figure 8. Illustration of the partial and complete Boolean learners on the same predicate matrix. In each table, rows correspond to vectors and columns correspond to clauses. A + indicates that the clause accepts a positive vector, and a − indicates that it rejects a negative vector. (a–c) show how the partial learner selects clauses until all negative clauses are rejected. In (a), the algorithm selects p_2 as it maximizes the weight and it discards positive vectors $\top\bot\bot$, $\top\bot\top$, $\bot\bot\bot$, and $\bot\bot\top$ because p_2 rejects them. (d–e) show how the complete learner adds larger clauses until all negative clauses are rejected. Any clause considered by the complete learner has to accept all positive vectors. Only clauses which accept all positive vectors are shown in this figure.

greedily chooses clauses to maximize the total weight of the positive vectors accepted by $\bigwedge C$. For each unselected clause, it calculates the total weight of the positive vectors accepted by the clause, and then chooses a clause c with the highest total. Any positive vectors rejected by c are discarded, and the weight totals for the unchosen clauses are recalculated. This continues until $\bigwedge C$ is sufficient to reject all negative clauses. Figures 8(a–c) illustrates the learner choosing clauses and discarding positive vectors until it finds a set C that rejects all negative vectors. ALIVE-INFER then uses the approximate set-cover algorithm from Figure 6 to find a subset of C that rejects all negative vectors.

In our prototype, ALIVE-INFER uses $K = 1$ and weights predicate vectors according to the number of associated positive examples. We plan to investigate the impact of these heuristics as future work. To increase the chances of finding an optimal formula, ALIVE-INFER may perform this algorithm several times, making different choices for the initial selected clause, and choosing the formula that accepts the most total weight.

4. Generalizing Concrete Expression DAGs

To further demonstrate the applicability of ALIVE-INFER, we generalize optimization patterns generated by Souper [21, 41], an LLVM IR–based superoptimizer. The initial prototype of Souper collects a database of expression DAGs that evaluate to either true or false [41]. It also generates concrete path conditions with such expression DAGs. We focus on expression DAGs without path conditions, because they can be translated to Alive. New peephole optimizations have been added to LLVM based on the patterns discovered by Souper [24]. In such scenarios, developers typically prefer to add a generalized version.

We create a generalized version of a Souper DAG by replacing all concrete constants in the source of the optimization with symbolic constants. However, we cannot replace a concrete constant in the target with a symbolic constant because Alive does not allow the definition of new symbolic constants in the target. Figure 9(1a) and Figure 9(2a) present the expression DAGs in Alive syntax. The generalized optimization and preconditions generated by ALIVE-INFER are shown in Figure 9(1b) and Figure 9(2b). To illustrate, the weakest precondition generated by ALIVE-INFER for the generalized optimization in Figure 9(1b) is

```
(C3 != 0 || C2 == 0) &&
(C2 u<= 1 || (C4 & ~C1) != 0 || C4 < 0) &&
((C4 & ~C1) != 0 || C4 >= C2) &&
(C3 != 0 || C1 == 0) &&
(C2 != 0 || C4 == 0 || (C4 & ~C1) !=0) &&
(C2 u> 1 || C4 u<= 1 || (C4 & ~C1) != 0) &&
(isSignBit(C4) || C2 + 1 >= 0 || (C4 & ~C1) != 0)
```

Unfortunately, this weakest precondition is not succinct. ALIVE-INFER also generated a partial precondition,

```
C4 & ~C1 != 0 && C3 != 0
```

also shown in Figure 9(1b). It is succinct and accepts 95% of the positive examples, which makes a case for generating partial preconditions.

5. Evaluation

We describe the ALIVE-INFER prototype, our methodology, and our experience inferring preconditions for LLVM peephole optimizations. Our experiments evaluate the effectiveness of the ALIVE-INFER prototype in generating both weakest and partial preconditions.

(1a) Alive translation for Souper pattern 512

```
%1 = and i32 1, %0
%2 = icmp eq 0, %1
%3 = xor 1, %2
%4 = icmp ne 0, %1
%5 = and %3, %4
%6 = or %5, %2
=>
%6 = 1
```

(1b) Generalized optimization

```
PInfer precondition:
((C4 & ~C1) != 0 &&  C3 != 0)
 %1 = and i32 C1, %0
 %2 = icmp eq C2, %1
 %3 = xor C3, %2
 %4 = icmp ne C4, %1
 %5 = and %3, %4
 %6 = or %5, %2
 =>
 %6 = 1
```

(2a) Alive translation for Souper pattern 537

```
%1 =srem i32 1, %0
%2 =lshr %1, 1
%3 =icmp ne 0, %2
=>
%3 = 0
```

(2b) Generalized optimization

```
PInfer precondition:
(C3 == 0 && (C1 u>> C2) == 0)
 %1 =srem i32 C1, %0
 %2 =lshr %1, C2
 %3 =icmp ne C3, %2
 =>
 %3 = 0
```

Figure 9. Generalization of optimization patterns generated by Souper with ALIVE-INFER. (a) Alive version of the Souper pattern. (b) Generalized optimization with all concrete values in the source replaced by symbolic constants along with the inferred precondition.

The* ALIVE-INFER *prototype. We built the ALIVE-INFER prototype by extending the publicly available Alive-NJ toolkit [33]. Alive-NJ also supports verification of floating point optimizations, but we leave precondition inference for those to future work.

ALIVE-INFER enhances Alive-NJ with three major features. (1) Implementations of the enumeration and learning algorithms, comprising roughly two thousand lines of Python code. (2) Safety analysis, which expresses the conditions under which an optimization target or precondition may have undefined behavior at compile-time. (3) Separation of the type checking and type assignment phases. ALIVE-INFER assigns each term an abstract type once during type checking or predicate enumeration. These abstract types are then mapped to concrete types during validation without the need of re-performing type checking.

In our experiments, we use Z3 4.4.1 [10] to handle SMT queries. The ALIVE-INFER prototype is open source and publicly available as part of the Alive-NJ toolkit.

Optimization suite. We use 417 optimizations from the Alive suite, a snapshot of optimizations from LLVM's InstCombine and InstructionSimplify passes. Some preconditions in Alive are weaker than LLVM's preconditions. Of these 417 optimizations, 195 require no precondition and 41 rely on dataflow analyses for runtime values. For the 195 optimizations that do not have a precondition, ALIVE-INFER successfully infers `true`. ALIVE-INFER does not support 41 optimizations that use dataflow analyses for runtime values. Of the remaining 181 that have a precondition in the Alive suite, seven require constant functions or predicates not currently supported by ALIVE-INFER. This leaves us with 174 optimizations for which ALIVE-INFER could possibly derive preconditions.

Methodology. In our experiments for precondition inference, we removed the precondition in the optimization and provided it to the ALIVE-INFER prototype. We compare the precondition generated by the ALIVE-INFER prototype and the original precondition for it in Alive (to determine if it is weaker or stronger). All experiments were performed on a 64-bit Intel Skylake–processor machine with four cores and 16 GB of RAM.

Precondition search To perform a fair assessment of the benefits of precondition inference using on-demand predicate learning, we created a variation of ALIVE-INFER that enumerates all possible preconditions until it finds a valid, weakest precondition. We refer to this method as precondition search, and call our modified prototype Alive-Search.

Alive-Search generates preconditions in CNF, using the predicate enumerator as a subroutine. A precondition's size is the sum of the sizes of its predicates. All preconditions of a given size are generated before any precondition of the next larger size.

Each precondition is tested against a set of examples generated using the method from Section 3.2. If the precondition accepts all positive instances and does not accept any negative instances, it is then verified by the SMT solver. If the solver finds counter examples, or additional positive examples which the precondition rejects, testing continues with the next precondition. It is never necessary to reconsider a previously-rejected precondition.

This algorithm has two advantages over precondition inference: it always finds a precondition of minimum size, and it can never get stuck with a difficult-to-separate sample. The disadvantage is that this algorithm cannot break the search problem into smaller parts. The number of preconditions and predicates for a given size both grow exponentially, with the number of preconditions growing somewhat faster. Consider an optimization for which the minimally-sized precondition has two predicates of sizes m and n. The number of preconditions which must be searched will be $O(c^{m+n})$, which is vastly larger than $O(c^m + c^n)$, the best-case amount of work needed for the predicate learner to find the two predicates. The exponential growth means that the predicate learner is still faster even if it wastes most of its effort learning predicates which will later be discarded.

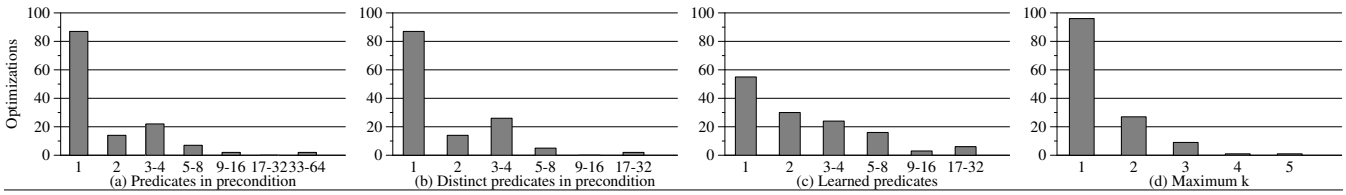


Figure 10. Information about weakest preconditions successfully inferred within 1000 s. The histograms show the number of optimizations with (a) the number of predicates in the precondition, (b) the number of distinct predicates in the precondition (a predicate and its negation are not considered distinct), (c) the number of predicates accepted by the learner during inference, and (d) the maximum number of predicates occuring in a disjunction (*i.e.*, the value of k reached by the Boolean formula learner).

5.1 Effectiveness in Generating Preconditions

We test the effectiveness of our approach by generating preconditions for the optimizations in the Alive suite. We ran inference for each optimization with a 1000-second timeout.

ALIVE-INFER successfully generated weakest preconditions for 133 out of the 174 optimizations. Although ALIVE-INFER could not generate weakest preconditions for the remaining 41 optimizations, it generated partial preconditions for 31 optimizations. For six optimizations, it was not able to learn sufficient predicates to generate any preconditions within the timeout period. In one case, the Boolean learner failed to find a formula. Finally, in three cases Z3 returned unknown during example generation or in final validation. In summary, ALIVE-INFER was able to generate either the weakest or a partial precondition for 164 out of the 174 optimizations.

Figure 10 provides summary information on the number of predicates in the generated weakest precondition, number of distinct predicates in the weakest precondition, the total number of predicates learned, and maximum disjunction size in the final formula learned by the Boolean learner. Figure 10(a) shows that about 80 optimizations in the suite have a single predicate in the precondition, and around 40 optimizations have two to four predicates. Figure 10(b) shows that the number of distinct predicates in the weakest precondition is lower than the number of predicates, which is common in formulae expressed in conjunctive normal form.

Figure 10(c) characterizes the number of predicates needed to separate the examples for various optimizations. ALIVE-INFER learned no more than 28 predicates for any optimization. Often, not all learned predicates were needed in the final formula. Learned predicates may occur multiple times in a precondition, so the maximum precondition size is greater than the largest number of learned predicates. Figure 10(d) shows that only 45 optimizations required disjunction to express a weakest precondition, indicated by having a clause size greater than one.

The number of predicates enumerated over the course of predicate learning varies widely. For nine optimizations where ALIVE-INFER found a weakest precondition, the learner considered more than ten thousand predicates and as many as 104,000. For nineteen, it considered between one and ten thousand. For sixteen, between a hundred and a thousand. For eighty-six, between eleven and a hundred. For four, between one and ten.

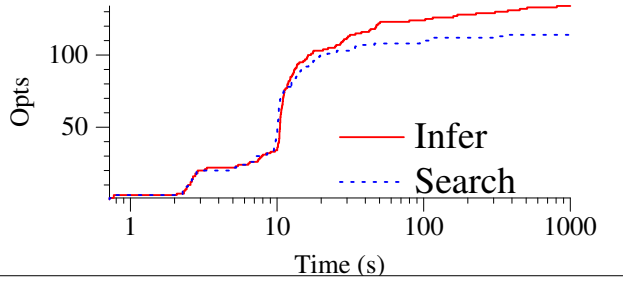


Figure 11. Number of optimizations for which a weakest precondition was inferred by precondition inference (Infer) and precondition search (Search) within a given time limit. The x-axis is running time, in seconds, used to infer the precondition of each optimization. The y-axis is the cumulative number of optimizations which required at most that time.

Preconditions with generalization. To evaluate the applicability of ALIVE-INFER with generalization, we translated 71 concrete optimization instances from the initial results of Souper to Alive and generalized them with symbolic constants. ALIVE-INFER was able to generate weakest preconditions for 51 optimizations. It generated partial preconditions for additional 3 optimizations. In the remaining 17 cases, Z3 hung while generating positive examples, Z3 returned unknown, or ALIVE-INFER could not learn a Boolean formula that rejects all negative vectors.

5.2 Comparison of ALIVE-INFER with Alive-Search

For comparison, we also experimented with precondition search (described earlier in this section) for the same set of optimizations. In contrast to ALIVE-INFER, Alive-Search was able to generate weakest preconditions for 114 of 174 optimizations. It performs similarly to ALIVE-INFER for optimizations with few predicates in the precondition. It times out for optimizations that have more than three predicates in the precondition.

Figure 11 reports the number of optimizations for which ALIVE-INFER and Alive-Search were able to generate weakest preconditions within a given amount of time. Both tools generate weakest preconditions in 10 seconds (per optimization) for about 100 optimizations. These optimizations have

one or two predicates in the precondition (see Figure 10(a)). In summary, we observe that predicate inference can find preconditions for more optimizations than predicate search within a given time limit.

5.3 Strength of Inferred Preconditions

We compare weakest preconditions generated by ALIVE-INFER with the preconditions in the Alive suite. We consider a precondition weakest if it accepts every example where the optimization is non-trivially valid; i.e., the source is well-defined for some run-time input. To determine whether a precondition generated by ALIVE-INFER (ϕ_I) accepts more examples than its counterpart in the suite (ϕ_A), we check the satisfiability of the formula $\phi_I \wedge \neg\phi_A$ using an SMT solver.

Of the 133 optimizations where ALIVE-INFER is able to generate a weakest precondition, 73 are weaker than the suite's precondition. Figure 12 shows four of these optimizations, with the preconditions from the suite and generated by ALIVE-INFER. For the remaining 61 optimizations, our prototype generated a precondition equivalent to its suite counterpart. Even the partial preconditions generated were often weaker than the suite preconditions. Of the partial preconditions generated, 15 are incomparable because there are examples which are accepted by the ALIVE-INFER precondition but not by the suite precondition and vice-versa. These include the example in Section 1.

The structure of LLVM's peephole optimization pass creates implicit assumptions for many optimizations: if two optimizations can apply to the same input program, only the first will be applied. ALIVE-INFER learns preconditions in isolation; assumptions must be explicit. Even taking these implicit assumptions into consideration, we have found optimizations where ALIVE-INFER finds a weaker precondition and reported them to LLVM's developers [35]. We plan to investigate others in the future.

6. Related Work

There is a large body of work on inferring specifications — preconditions, postconditions, and invariants — for general purpose programs [1, 3, 4, 6, 9, 11–15, 39, 43, 46, 47]. Data-driven approaches have also been explored for inferring specifications [13–15, 39, 43]. We primarily focus on closely related work in this section.

PIE. Our work is inspired by PIE [39], which generates preconditions for general-purpose programs. PIE uses predicate learning, which it calls feature learning, along with a Boolean learner to separate positive and negative examples. ALIVE-INFER differs from PIE by addressing challenges specific to LLVM and Alive. First, we identify the need for succinct partial preconditions and propose a weighted partial Boolean formula learner. Second, we propose a strategy to generate positive and negative examples while handling polymorphic types and compile-time undefined behavior. Third, we design a predicate learner which can reason about predicates with potential compile-time undefined behavior.

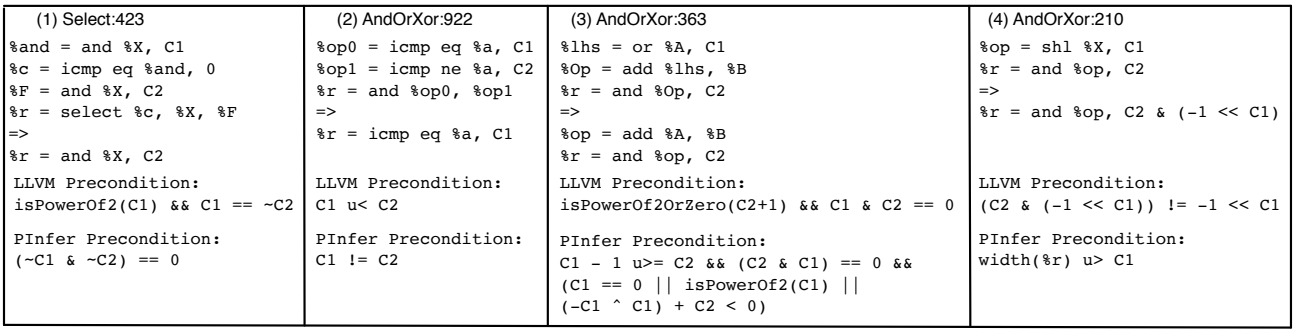
Compiler precondition synthesis. Prior approaches have also explored precondition generation for compiler optimizations [7, 31, 44]. PSyCO [31] synthesizes read-write preconditions given a finite predicate set. They do not address the complexities of bitvector arithmetic and the interaction with undefined behavior. Optgen [7] automatically generates all peephole optimizations within a specified size bound and verifies their correctness. These optimizations may include preconditions, which are expressions of the form `expr == 0` and are found using enumeration.

Logical abduction methods. Another approach to precondition inference is logical abduction [11, 16]. Methods using quantifier elimination [11] are promising, but methods for eliminating quantifiers in bitvector algebra work only for a small subset of operations [19]. We initially tried logical abduction methods by restricting optimizations to use only linear integer arithmetic (LIA) but settled on a data-driven approach to increase its applicability.

Data-driven inference methods. Other prior data-driven approaches often work only with predefined predicates [15, 43, 47]. Researchers have used counter-example guided refinement [8], similar to ALIVE-INFER, by beginning with overlapping positive and negative sets and refining them by finding counter-examples [46]. They also require a fixed set of predefined predicates. ICE and ICE-DT [13, 14] use positive, negative, and implication examples for synthesizing invariants. They use either a template-based synthesis or a decision tree learning algorithm to generate invariants using a fixed set of attributes. ALIVE-INFER, similar to PIE, learns and synthesizes predicates on-demand.

Search techniques and superoptimization. ALIVE-INFER's inference can be viewed as a variant of various symbolic, stochastic, and enumerative search strategies employed in program synthesis [2, 17, 22, 48, 50] and superoptimizers [5, 20, 32, 40, 45]. ALIVE-INFER can be used to generalize/validate patterns generated by superoptimizers.

Compiler correctness. A compiler can be written in a mathematical theorem prover (*e.g.*, CompCert [27], Vellvm [52, 53]), which would require one to figure out the specification in such a setting [37, 49]. Alternatively, various other DSLs have also been proposed for compiler construction [23, 26]. ALIVE-INFER generates preconditions or optimizations expressed in Alive [30]. In principle, ALIVE-INFER can apply to other DSLs. Our recent work has explored compiler non-termination errors with a suite of peephole optimizations [34], which typically occurs when profitability metrics are not included in the precondition. The weakest preconditions inferred by ALIVE-INFER should be checked with those tools before including them in LLVM to avoid non-termination errors.



(1) Select:423

```
%and = and %X, C1
%c = icmp eq %and, 0
%F = and %X, C2
%r = select %c, %X, %F
=>
%r = and %X, C2
```

LLVM Precondition:
`isPowerOf2(C1) && C1 == ~C2`

PInfer Precondition:
`(~C1 & ~C2) == 0`

(2) AndOrXor:922

```
%op0 = icmp eq %a, C1
%op1 = icmp ne %a, C2
%r = and %op0, %op1
=>
%r = icmp eq %a, C1
```

LLVM Precondition:
`C1 u< C2`

PInfer Precondition:
`C1 != C2`

(3) AndOrXor:363

```
%lhs = or %A, C1
%Op = add %lhs, %B
%r = and %Op, C2
=>
%op = add %A, %B
%r = and %op, C2
```

LLVM Precondition:
`isPowerOf2OrZero(C2+1) && C1 & C2 == 0`

PInfer Precondition:
`C1 - 1 u>= C2 && (C2 & C1) == 0 &&`
`(C1 == 0 || isPowerOf2(C1) ||`
`(-C1 ^ C1) + C2 < 0)`

(4) AndOrXor:210

```
%op = shl %X, C1
%r = and %op, C2
=>
%r = and %op, C2 & (-1 << C1)
```

LLVM Precondition:
`(C2 & (-1 << C1)) != -1 << C1`

PInfer Precondition:
`width(%r) u> C1`

Figure 12. A sample of optimizations where ALIVE-INFER generated a weaker precondition compared to the precondition in LLVM/Alive. We provide the name of the optimization in the Alive suite, the LLVM/Alive precondition and the ALIVE-INFER precondition. (1) Consider the instance `C1 = 3, C2 = 14` for 4-bit integers, *i.e.*, `0011` and `1110`. These satisfy neither of the clauses in LLVM's precondition, but do satisfy the ALIVE-INFER's precondition, which can be rewritten as `C1 | C2 == -1`. (2) This optimization's source calculates $a = C_1 \wedge a \neq C_2$ and the target $a = C_1$. By the transitive property, this is equivalent to $a = C_1 \wedge C_1 \neq C_2$. ALIVE-INFER generates the equivalent precondition `C1 > C2 || C1 < C2`. (3) Consider the instance `C1 = 10, C2 = 2` for 4-bit integers, *i.e.*, `1100` and `0010`. This is rejected by LLVM's precondition, because three is not a power of two, but is accepted by ALIVE-INFER's. (4) ALIVE-INFER's precondition is clearly weaker, as it will accept the cases where `C2` is masked by `-1 << C1` as long as `C1` is less than the bit width. For example, `C1 = 2, C2 = 14` for 4-bit integers, *i.e.*, `0001` and `1110`.

7. Conclusion

We show that it is possible to infer preconditions for peephole optimizations in LLVM using a data-driven approach with on-demand predicate learning. We highlight the trade-off between applicability and succinctness of the precondition. The ALIVE-INFER prototype addresses the challenges of polymorphic types and compile-time undefined behavior in the precondition language to generate both weakest and succinct partial preconditions. Our goal is to assist LLVM developers in debugging an invalid optimization. ALIVE-INFER is likely to be useful to LLVM developers, as it is able to generate preconditions weaker than LLVM's preconditions.

Acknowledgments

We thank Aarti Gupta, Adarsh Yoga, Jay Lim, and the PLDI reviewers for their feedback on drafts of this paper. We thank John Regehr for his blog posts on Souper. This paper is based on work supported in part by NSF CAREER Award CCF–1453086, a sub-contract of NSF Award CNS–1116682, a NSF Award CNS–1441724, a Google Faculty Award, and gifts from Intel Corporation.

References

[1] A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal Specification Synthesis. In *Proceedings of the 43rd Annual Symposium on Principles of Programming Languages*, POPL, pages 789–801, Jan. 2016.

[2] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-Guided Synthesis. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design*, FMCAD, pages 1–17, Oct. 2013.

[3] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 98–109, Jan. 2005.

[4] G. Ammons, R. Bodík, and J. R. Larus. Mining Specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 4–16, 2002.

[5] S. Bansal and A. Aiken. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 394–403, Oct. 2006.

[6] M. Barnett and K. R. M. Leino. Weakest-precondition of Unstructured Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE, pages 82–87, Sept. 2005.

[7] S. Buchwald. Optgen: A Generator for Local Optimizations. In *Proceedings of the 24th International Conference on Compiler Construction*, CC, pages 171–189, 2015. ISBN 978-3-662-46663-6.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. CAV, 2000.

[9] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic Inference of Necessary Preconditions. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI, pages 128–148, Jan. 2013.

[10] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, pages 337–340, 2008.

[11] I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive Invariant Generation via Abductive Inference. In *Proceedings of the*

2013 ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA, pages 443–456, Oct. 2013.

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, Dec. 2007.

[13] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A Robust Learning Framework for Synthesizing Invariants. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV, pages 69–87, July 2014.

[14] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning Invariants using Decision Trees and Implication Counterexamples. In *Proceedings of the 43rd Annual Symposium on Principles of Programming Languages*, POPL, pages 499–512, Jan. 2016.

[15] T. Gehr, D. Dimitrov, and M. T. Vechev. Learning Commutativity Specifications. In *Proceedings of the 27th International Conference on Computer Aided Verification*, CAV, pages 307–323, July 2015.

[16] R. Giacobazzi. Abductive analysis of modular logic programs. In *Proceedings of the 1994 International Symposium on Logic programming*, ISPL, pages 377–391, Nov. 1994.

[17] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of Loop-free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, June 2011.

[18] Y. Jiang. [Patch]InstCombine pattern for ICMP. `http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20140818/231300.html`, 2014. Retrieved 2016-11-10.

[19] A. K. John and S. Chakraborty. Quantifier Elimination for Linear Modular Constraints. In *Proceedings of the 4th International Congress on Mathematical Software*, ICMS, pages 295–302, Aug. 2014.

[20] R. Joshi, G. Nelson, and Y. Zhou. Denali: A practical algorithm for generating optimal code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):967–989, Nov. 2006.

[21] J. Ketema, J. Regehr, J. Taneja, P. Collingbourne, and R. Sasnauskas. A superoptimizer for LLVM IR. `https://github.com/google/souper`. Retrieved 2016-11-14.

[22] A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-Based Model Checking for Recursive Programs. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV, pages 17–34, July 2014.

[23] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 327–337, 2009.

[24] C. Lattner. ComputeMaskedBits and friends should know that multiplying by a power of two leaves low bits clear. `https://bugs.llvm.org//show_bug.cgi?id=19711`, 2016. Retrieved 2017-03-14.

[25] V. Le, M. Afshari, and Z. Su. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 216–226, 2014.

[26] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 364–377, 2005.

[27] X. Leroy. A Formally Verified Compiler Back-end. In *Journal of Automated Reasoning*, 2009.

[28] C. Liam. [Patch]Implementing a proposed InstCombine optimization. `http://lists.llvm.org/pipermail/llvm-dev/2016-April/098104.html`, 2016. Retrieved 2016-11-10.

[29] N. Lopes. RFC: Killing undef and spreading poison. `http://lists.llvm.org/pipermail/llvm-dev/2016-October/106182.html`, 2016. Retrieved 2016-11-10.

[30] N. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 22–32, 2015.

[31] N. Lopes and J. Monteiro. Weakest Precondition Synthesis for Compiler Optimizations. In *Proceedings of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI, pages 203–221, 2014.

[32] H. Massalin. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126, 1987.

[33] D. Menendez and S. Nagarakatte. Alive-NJ. `https://github.com/rutgers-apl/alive-nj`. Retrieved 2016-04-16.

[34] D. Menendez and S. Nagarakatte. Termination-Checking for LLVM Peephole Optimizations. In *Proceedings of the 38th International Conference of Software Engineering*, ICSE, pages 191–202, May 2016.

[35] D. Menendez and S. Nagarakatte. Weaker (more general) precondition for bit-tests in InstructionSimplify. `http://lists.llvm.org/pipermail/llvm-dev/2017-March/111000.html`, 2017. Retrieved 2017-03-14.

[36] D. Menendez, S. Nagarakatte, and A. Gupta. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *Proceedings of the 23rd Static Analysis Symposium*, pages 317–337, 2016.

[37] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman. Verified Peephole Optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 448–461, June 2016.

[38] A. Nötzli and F. Brown. LifeJacket: Verifying precise floating-point optimizations in LLVM. `http://arxiv.org/pdf/1603.09290v1.pdf`, 2016. Retrieved 2016-04-04.

[39] S. Padhi, R. Sharma, and T. Millstein. Data-driven Precondition Inference with Learned Features. In *Proceedings of the 37th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16, pages 42–56, 2016.

[40] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling Up Superoptimization. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 297–310, Apr. 2016.

[41] J. Regehr. Early Superoptimizer Results. `http://blog.regehr.org/archives/1146`. Retrieved 2016-11-14.

[42] J. Regehr. Signed Division and InstCombine. `http://lists.llvm.org/pipermail/llvm-dev/2016-June/100375.html`, 2016. Retrieved 2016-11-10.

[43] S. Sankaranarayanan, S. Chaudhuri, F. Ivančić, and A. Gupta. Dynamic Inference of Likely Data Preconditions over Predicates by Tree Learning. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 295–306, 2008.

[44] E. R. Scherpelz, S. Lerner, and C. Chambers. Automatic Inference of Optimizer Flow Functions from Semantic Meanings. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 135–145, June 2007.

[45] E. Schkufza, R. Sharma, and A. Aiken. Stochastic Superoptimization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 305–316, 2013.

[46] M. N. Seghir and D. Kroening. Counterexample-Guided Precondition Inference. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP, pages 451–471, Mar. 2013.

[47] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A Data Driven Approach for Algebraic Loop Invariants. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 574–592, 2013.

[48] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial Sketching for Finite Programs. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, Oct. 2006.

[49] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *PLDI '10: Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, 2010.

[50] E. Torlak and R. Bodik. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 530–541, June 2014.

[51] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 283–294. ACM, 2011.

[52] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 427–440, 2012.

[53] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal Verification of SSA-Based Optimizations for LLVM. In *ACM SIGPLAN 2013 Conference on Programming Language Design and Implementation*, 2013.