PRACTICAL FORMAL TECHNIQUES AND TOOLS FOR DEVELOPING LLVM'S PEEPHOLE OPTIMIZATIONS

by

DAVID MENENDEZ

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Santosh Nagarakatte

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

January, 2018

ABSTRACT OF THE DISSERTATION

Practical formal techniques and tools for developing LLVM's peephole optimizations

by David Menendez

Dissertation Director: Santosh Nagarakatte

Modern compilers perform extensive transformation of code in order to optimize running time and binary code size. These occur in multiple passes, including translations between representations at different levels of abstraction and transformations which restructure code within a particular representation. Of particular interest are optimizations that operate on a compiler's intermediate representation (IR), as these can be shared across programming languages and hardware architectures. One such optimization pass is LLVM's peephole optimizer, which is a suite of hundreds of small algebraic transformations which simplify code and perform canonicalization. Performing these transformations not only results in faster software, but simplifies other optimization passes by reducing the number of equivalent forms they must consider.

It is essential that these optimizations preserve the semantics of input programs. Even a small transformation which changes the value computed by a code fragment or introduces undefined behavior can result in executable programs with incorrect or unpredictable behavior. Optimizations, and analysis of optimizations, must be particularly careful when treating undefined behavior, as modern compilers increasingly use the knowledge that certain operations are undefined in order to streamline or eliminate

code—occasionally in ways that are surprising to compiler users. Unfortunately, compiler developers can also overlook undefined behavior or fail to consider rare edge cases, resulting in incorrect transformations. In particular, LLVM's peephole optimizer has historically been one of the buggier parts of LLVM.

To aid the development of correct peephole transformations in LLVM, we introduce Alive, a domain-specific language for specifying such transformations. Selecting a small yet expressive subset of LLVM IR allows for automated verification of Alive transformations, and the Alive toolkit can generate an implementation of a correct transformation suitable for inclusion in LLVM. The correctness checks for Alive consider the various forms of undefined behavior defined by LLVM and ensure that transformations do not change the meaning of a program. Alive specifications can include a mixture of integer and floating-point operations, and transformations can be generalized over different types.

Some transformations require a precondition in order to be correct. These preconditions may be simple, but occasionally it is challenging to find a precondition that is sufficiently strong while remaining widely applicable. To assist in this process, the Alive toolkit includes Alive-Infer, a data-driven method for synthesizing preconditions. Depending on the complexity of the transformation, the weakest precondition sufficient to make a transformation correct may not be desirable, so Alive-Infer can provide a choice of concise but stronger preconditions. The Alive-Infer method automatically finds positive and negative examples to guide inference and finds useful predicates through enumeration.

Finally, specifying transformations in Alive enables analyses of multiple transformations and their interaction. It is possible to have transformations or sequences of transformations which can be applied indefinitely to a finite input. This dissertation presents a method for testing whether such a sequence can be applied indefinitely to some input.

Alive demonstrates that a properly chosen abstraction can provide the benefits of formal code verification without the need for manually written proofs, and can enable new techniques and analyses to assist development.

Acknowledgements

This dissertation could not have been written without the assistance of many people. Foremost among them is my advisor, Santosh Nagarakatte, who brought me into the Alive project, helped develop my ideas, and read several of my drafts.

Alive itself is a collaboration with Nuno Lopes and John Regehr, who provided key insights and experience with LLVM semantics and SMT solving. Aarti Gupta assisted our investigation of ways to integrate floating-point reasoning into Alive.

The members of my doctoral committee, Rajeev Alur, Ulrich Kremer, and Thu Nguyen, helped refine my presentation of ideas and suggested further areas of exploration.

My colleagues in the Rutgers Architecture and Programming Languages Research Group, especially Adarsh Yoga and Jay Lim, provided support and feedback throughout the development of Alive.

In addition to the support of Rutgers University, the Alive project was funded by the National Science Foundation, a Google Faculty Award, and gifts from Intel Corporation.

Finally, this dissertation could not have been completed without the assistance and patience of my wife, Lisa, and my daughter, Alexandra.

Dedication

In memory of my father, Ronald Menendez.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Compilers are essential tools for software development, providing a bridge between programmer-friendly high-level languages and the simpler, low-level instruction sets used by computer processors. Modern compilers are large, sophisticated systems translating source code through several intermediate representations before finally producing executable software. An important part of this process is optimization, where the compiler rewrites a fragment of code into a more efficient form while preserving its semantics.

Compilers should be trustworthy. A buggy compiler may change the semantics of a program as it compiles it, producing an executable that behaves incorrectly, even when the source program is correct. Errors caused by compiler bugs are particularly frustrating for software developers to diagnose, as they must first determine that the fault is not in their own code. Worse yet, the error may appear or disappear depending on the version of the compiler being used, or may only occur for certain semantically equivalent forms of a program.

The complexity of modern compilers provides many opportunities for errors. Mainstream compiler systems such as GCC and LLVM support multiple languages and dialects and target hundreds of popular and obscure machine architectures. Input languages such as C and C++ have highly complex semantics and sophisticated features that can interact in surprising ways. Architectures such as Intel's X86 series include hundreds of different instructions, each of which may have side effects such as setting condition codes in addition to their primary function. Language standards evolve, and new architectures are created. Between the source and target languages, compilers have multiple intermediate representations and optimization passes. Mistakes may occur at

any of these points, due to misunderstood semantics, overlooked boundary conditions, or simply incorrect implementation.

Indeed, a look at the bug trackers for GCC and LLVM will show multiple open compiler bugs. These range from formatting errors and suggestions for improvements, to compiler crashes, to miscompilation. Sometimes, a claim that a compiler has produced an incorrect program stems from a misunderstanding of the programming language's semantics, but other times this reflects a genuine error in the compiler.

Many methods have been developed to detect or prevent compiler bugs. Some, such as testing, are simple to implement but limited in the guarantee they provide. Others provide very strong guarantees, but are cumbersome to implement. Our goal is to develop methods that are easy to use and provide strong guarantees.

1.1 Making Compilers Robust

Compiler developers employ several methods for detecting bugs. Most compiler projects will require some assurance before accepting new code. This may involve a manual inspection of the new code or the creation of test cases to demonstrate correctness. Compiler projects include extensive test suites that are run frequently, using the latest in-development versions of the compiler, in order to detect recently introduced problems.

LLVM, for example, has two major test suites. The regression tests check individual features for correctness and test for specific bugs by compiling several thousand short code fragments. These are expected to always pass, and are performed whenever changes are committed. When a new bug is found, one or more regression tests are created to ensure it is not reintroduced later. LLVM's larger test suite, sometimes called the nightly tests, comprise several hundred complete programs written in C and C++. The programs are compiled and executed with specific inputs, and their outputs are compared with expected outputs. These tests not only check whether LLVM has compiled the programs correctly, but also measure the efficiency of compilation and the efficiency of the compiled programs.

While these methods are effective at catching bugs, they have limits. It is difficult to create a test suite that exercises all execution paths. Some bugs may arise from the interaction between different components only when certain compilation options are chosen. Testing code generation requires test machines for every target hardware platform, or cross-compilation support (which may itself be a source of bugs). It is even conceivable that a bug could depend on the compiler used to compile the compiler.

1.1.1 Random testing

One way to address the problems of creating large numbers of test cases and increasing coverage of the compiler is to randomly generate test cases [15, 49, 100, 107]. A well-designed test generator can generate any desired number of test cases distributed uniformly among some space of possible programs. These are especially useful if they meet some minimal criteria for being interesting (*e. g.*, having valid syntax or not having undefined behavior) without being biased towards program patterns common in existing software or test cases. This increases the chances that a test case will involve combinations of features not found in existing test suites.

One challenge when randomly generating an input program, compiling it, and observing its execution, is determining whether its behavior is correct. One possible method is to generate programs that behave in a particular way and check whether that behavior is observed. This method works well when testing specific features, such as calling conventions [73] or arithmetic expressions [91, 92], where the correct behavior is easily determined through other means.

Another approach is *differential testing* [33, 82, 113, 129], which does not require foreknowledge of a test program's behavior. Instead, the test program is compiled multiple times using different compilers. If the program is deterministic, the compiled versions should all have the same behavior. Differences in behavior among the compiled programs indicate that one or more of the compilers did not preserve the meaning of the test program. The behavior seen in the majority of compiled programs is most likely to be correct, as the chances that a majority of compilers introduce the same error is small.

CSmith [127] is a differential testing system for C compilers that generates programs using multiple language features while avoiding programs whose behavior depends on undefined or unspecified aspects of C. As it is impractical to avoid certain forms of undefined behavior, CSmith uses a combination of static and dynamic checks to ensure that the generated test programs have only one expected behavior. CSmith has found more than 400 bugs in GCC, LLVM, and other tools.

Instead of generating entirely new programs, it is also possible to create new programs by modifying existing programs. Using the idea of *equivalence modulo inputs* (EMI), we can define a class of programs that have identical behavior for inputs within a particular set. For example, given a set of inputs I, we can define a set of programs such that, for every input $i \in I$, all programs in the set will have the same behvior when given i. However, two programs P and Q whose behaviors coincide for inputs in I may have different behaviors for inputs not in I. When compiling P and Q, the compiler must consider their behavior for all possible inputs, which may exercise different parts of the compiler, as different static analyses will lead to different optimizations being performed. The compiled versions of P and Q should still behave identically for inputs in I, but it is possible that the differences in compilation will cause an incorrect transformation to be applied to one but not the other, leading to divergent behavior for some input in I.

Orion [64] uses EMI by observing the execution of a program P on inputs in I and noting which fragments of P are not executed for any input in I. Orion generates variants of P by randomly removing some of these unused fragments. Since these deleted fragments were unused for the inputs in I, the variants should have the same behavior as P for all inputs in I. However, these smaller variants may trigger different optimizations, such as inlining or loop invariant motion, which can lead to different behavior if the newly-triggered optimizations have bugs. Orion has been used to find more than 1500 bugs in GCC, LLVM, CompCert, and other tools.

1.1.2 Formal correctness proofs

The primary limitation of any testing system is the impracticality of testing all possible behaviors. A higher level of assurance is provided through the use of formal methods to prove the correctness of code.

A proof of correctness for a program has three parts. First, the desired property that the program should have must be specified. This will be a logical statement about the program's behavior (*e. g.*, all allocated resources are deallocated exactly once). Second, the behavior of the program must be determined. This will typically involve a formally specified semantics for the language used to write the program, which can then be used to describe the behavior of specific programs in a logical notation. Finally, there must be a formal proof that the behavior of the program has the desired property.

Correctness of compilers

Compilers are a natural subject of formal correctness proofs, as they are not only large software projects themselves, but part of the chain of trust for the programs compiled with them. Compilers can be more challenging to prove correct than other programs. A correctness proof effectively must reason about three programs: the compiler itself, the program being compiled, and the result of compilation. Each program may be given in a different language, with semantics specified at a different level of abstraction.

A compiler can be modeled as a function $Comp$ that maps an input program S to either a compiled program C or an error condition, indicating that compilation failed (*e. g.*, because S has a syntax error). We will write $Comp(S) = \mathsf{OK}(C)$ to indicate a successful compilation.

A typical correctness condition for a compiler requires that some semantic property of a source program S be preserved in its corresponding compiled program C. A strong semantics preservation property is *bisimulation*, in which a behavior is possible for C if and only if it is possible for S. Bisimulation is too strong to be used for languages with any non-determinism, such as the order of evaluating subexpressions in C. A weaker property is preservation of a specification, where the behaviors of C must meet some

specification if all behaviors of S do. We will write $S \approx C$ to indicate that C preserves the semantics of S according to a chosen property.

There are several methods for proving a compiler to be correct. A *verified compiler* has a formal proof that $Comp(S) = \mathsf{OK}(C) \implies S \approx C$, meaning successful compilation produces a compiled program that preserves the semantics of the source program. This provides a very high level of assurance, but the complexity of modern compilers and their input and output language semantics means it is often not feasible to write such a proof.

An alternative approach is translation validation [93, 99, 105]. In addition to the compiler, we provide a separate *validator*, which computes a predicate $Valid(S, C)$ that holds when $S \approx C$. Because S and C are specific, concrete programs, it is possible to perform validation using methods such as abstract interpretation and model checking [32]. A *verified validator* accompanies the validator with a formal proof that $Valid(S, C) \implies S \approx C$. An unverified compiler equipped with a verified validator provides as much assurance as a verified compiler, if it only returns validated output programs and reports an error when validation did not succeed. Naturally, the disadvantage of this approach is that validation must be performed each time a program is compiled.

CompCert [68–70, 90, 120] is a C compiler written using the proof assistant Coq [23]. Its input and output languages, and each of its multiple intermediate representations, have formally specified semantics. The functions that translate from one representation to the next each are verified or include a verified validator. Coq ensures that the proof both reflects the implementation of the function and is itself valid. This is a monumental achievement, requiring many programmer-years of proof development, and results in a compiler with a very high assurance of correctness. For this reason, CompCert is popular with embedded systems developers.

One disadvantage of this approach is the difficulty of extending the compiler. Adding support for new platforms or optimizations requires further proof engineering by expert developers. Changes to the language standards may require extensive re-engineering, if

they result in changed semantics. Nevertheless, efforts have been made to simplify extending CompCert, using techniques such as verifiable domain-specific languages [120].

Languages for automated correctness checking

Even if fully proving the correctness of a compiler is impractical, it is still useful to prove the correctness of individual passes within a compiler. The transformations performed by a single pass may fit a more specific pattern than the compiler as a whole, simplifying the reasoning that must be done.[1] For optimization passes, the input and output languages are the same, simplifying the semantics preservation properties.

Further simplification can be achieved by changing how the transformations are written. Instead of writing transformations in a general-purpose programming language, with its corresponding complex semantics, transformations can be specified in a *domain-specific language* (DSL) designed to express a class of transformations and simplify reasoning about their correctness [46, 62, 66, 67, 79, 125]. The challenge to designing such a DSL is making it flexible enough to specify many transformations while keeping it simple enough for automated or semi-automated correctness checking.

Unlike verified code in an environment like Coq, the proof of correctness for a DSL-specified transformation is not necessarily checked against its implementation. Thus, a transformation may be correct by specification, but a mistake in the implementation—written in a different language, possibly by different developers—would render this verification meaningless. This vulnerability can be reduced by designing the language so that an implementation can be automatically generated from the specification, thus ensuring that the implementation and specification are kept in sync. A tool that proves the correctness of a transformation and produces an implementation is an example of a *verifying compiler* [52], which checks the correctness of its input programs.

---

[1] CompCert is divided into separately verified stages for this reason.

```
int foo(int x) {
  int z = x * 51;
  return z / 3;
}
```

(a) Source program

```
define i32 foo(i32 %x) {
  %z = mul nsw i32 %x, 51
  %r = sdiv i32 %z, 3
  ret i32 %r
}
```

(b) IR translation

```
define i32 foo(i32 %x) {
  %r = mul nsw i32 %x, 17
  ret i32 %r
}
```

(c) After optimization

Figure 1.1: Peephole optimizations perform local optimizations, such as transforming the code on the left into the code on the right.

1.2 Peephole Optimizations

Peephole optimizations are local transformations of intermediate representation (IR) code. They perform algebraic simplification and enable other optimizations by rewriting code into a normal form. Individually, peephole optimizations provide only small improvements to code, but they are typically applied repeatedly, until the code reaches a steady state, and one optimization may create an opportunity for another. In this way, complex operations can be transformed and simplified.

Figure 1.1 shows a typical application of a peephole optimization. The input C program in Figure 1.1(a) is first translated into IR, represented by the LLVM IR code in Figure 1.1(b). Next, the peephole optimizer recognizes a pattern in the program: a multiplication by a constant a followed by a division by a constant b, where a is a multiple of b, *i. e.*, $a = bc$ for some integer c. These operations can safely be replaced with a multiplication by c, which will compute the same value, as shown in Figure 1.1(c).

The need for a to be a multiple of b comes from the fact that these are integer operations, but these are fixed-width, two's-complement machine integers, not mathematical integers. We also must consider what happens if the input is too large and the first multiplication overflows. This would result in a negative value in the original program, but (potentially) a positive value in the optimized program. So is this optimization unsound?

As it happens, C declares any operation on signed integers that overflows to have undefined behavior. Thus, a compiler would be free to assume that the multiplication in Figure 1.1(a) never overflows. This is reflected in the IR translation in Figure 1.1(b): the `nsw` attribute (or, "no signed wrap") on the `mul` instruction promises that the result

will fit in 32 bits. Therefore, we can ignore what happens when the multiplication overflows, because no particular behavior is required in those cases.

This example illustrates some of the challenges of writing correct peephole optimizations. The semantics of LLVM's integer types seem simple at first glance, but it is easy to confuse the rules for modular integer arithmetic with regular integer arithmetic, and it is even easier to lose sight of how LLVM handles undefined behavior.

1.3 Problem Statement

We want a method for creating peephole optimizations that are provably correct. Furthermore, we want a method that is easy to use. Ideally, developers should be able to reason directly about the structures being manipulated—IR instructions, in this case—instead of being concerned with the specific representation of those structures or the formal semantic interpretation of those structures. Thus, we propose a domain-specific language for specifying peephole optimizations.

Now our question becomes: *How can we design a language to enable automated verification of transformations?* In general, it is impossible to prove that an arbitrary program is correct for all inputs, so we must restrict the expressiveness of our language to something that we can reason about. We should also restrict our language to be constructive: a transformation should indicate a specific target for the transformation, rather than providing constraints that could be fulfilled in several ways.

1.4 Overview of Alive

We develop a domain-specific language, Alive, for verifying LLVM IR peephole optimizations. Its constructs are instructions and other IR values, but generalized to the abstraction level used by LLVM's peephole optimizer. (For example, values in LLVM IR have concrete types, but peephole optimizations are typically generic over types.) Alive optimizations include a source, which describes what code patterns the optimization applies to, a target, which describes the new code that replaces the source, and preconditions, which prevent applying the optimization when it would be invalid or otherwise

undesirable. The source and target are expressed as LLVM IR–like instructions, which may include uninterpreted variables. Their semantics are designed to match the code fragments to which they apply, and must include the values computed by the instructions and the conditions where those values are undefined. Alive optimizations also have operational semantics that describe the specific transformations applied to LLVM's IR data structures.

Alive assists developers by automating the process of checking an optimization's correctness and ensuring its implementation meets its specification. What other assistance can we offer to optimization developers? Consider a developer who has found a seemingly desirable optimization that Alive deems invalid. This optimization may be wrong in all circumstances, or it may be valid to apply in some, but not in others. Building on Alive, we create a tool that can detect whether an optimization is ever valid and infer preconditions that indicate those circumstances where it is valid. We also support the converse situation, where an optimization has a precondition that is too strong, preventing the optimization from applying in situations where it would be valid. Using our tool, a developer can find weaker preconditions that still ensure correctness.

Finally, we examine whether the validity of individual optimizations is sufficient to show the correctness of a compiler. Is it possible to add a correct optimization to a correct compiler and create an incorrect compiler? One problem that can arise involves non-termination in the peephole optimizer. Peephole optimization typically proceeds until no further opportunities for optimization remain. If an optimization can apply to its own output, or if one optimization undoes the work of another, then the compiler may never reach a state where no further optimization can be performed. That is, compilation of certain programs may run indefinitely without producing a result. In a sense, producing no output is less serious than producing incorrect output, but both are undesirable and difficult for compiler users to diagnose and fix. Building on Alive, we develop a method for determining whether a sequence of optimizations can be performed indefinitely on a finite input. Using it, we are able to search collections of optimizations for potential interactions between optimizations that could lead to non-termination.

```
Pre: C2 != 0 && C1 % C2 == 0
  %m = mul nsw %X, C1
  %r = sdiv %m, C2
=>
  %r = mul nsw %X, C1 / C2
```

Figure 1.2: An Alive specification for the optimization demonstrated in Figure 1.1

These three applications of Alive—transformation verification, precondition inference, and non-termination detection—are discussed in more detail in the following sections.

1.4.1 Verifying Transformations

Our language for specifying LLVM IR peephole optimizations, Alive, is designed to resemble the IR. It relaxes some requirements of the IR for generality, while restricting its coverage to instructions that are amenable to automated reasoning: primarily arithmetic and logical operators, comparisons, and conversions.[2] It does not address control flow instructions, such as branches and `phi` nodes, thus avoiding the need to reason about loops.

Figure 1.2 shows an Alive specification for the peephole optimization discussed in Section 1.2. It has three parts: a *precondition*, which limits when the optimization may apply, a *source pattern*, which indicates the form of IR code where the optimization may transform, and a *target*, which indicates the result of the transformation. Compare the source and target with the LLVM IR code shown in Figures 1.1(b) and 1.1(c), respectively.

Alive generalizes LLVM IR in several ways. First, it is parametric over types. While IR programs have fixed, concrete types, peephole optimizations usually do not require their input to have specific types. An algebraic rewrite is generally the same, no matter the width of the integers involved. Second, compile-time constants may be left abstract. `C1` and `C2` are examples of *symbolic constants*, which correspond to values that will be known at compile time, but are not restricted to specific concrete values. Because they

[2]Some implementations of Alive have limited support for basic memory operations.

are known at compile time, the precondition and target of the optimization may use them to compute new values and make judgements.

Alive optimizations are verified through translation into first-order logical propositions augmented with operations and predicates over fixed-width integers and floating-point values. These formulae are constructed so that they are satisfiable if and only if the optimization is invalid, having the general form "the precondition is satisfied and the source and target compute different values". This formula is passed to a solver for Satisfiability Modulo Theories (SMT), which determines whether any value assignment to the variables and symbolic constants satisfies the formula. If no such assignment exists, the optimization is valid. Otherwise, the assignment generated by the solver can be reported to the user as a counter-example for the optimization.

There are some additional points that must be considered when verifying Alive optimizations. First, Alive optimizations may be type parametric. Reassociating two integer addition instructions, for example, is valid regardless of the bit width of the instructions involved. In contrast, SMT formulae must have fixed types. Unfortunately, there appears to be no general way to use the correctness of an optimization at one type assignment to show its correctness at another. Thus, to prove an optimization is correct, it must be verified separately for each type assignment.[3] Second, Alive optimizations must correctly handle LLVM instructions that have undefined behavior for some inputs. In particular, the target of an optimization must never introduce undefined behavior for an assignment of variables where the source is well-defined. For those assignments where the source has undefined behavior, any behavior in the target is considered valid.

Third, we must ensure that applying the optimization cannot crash the compiler. If any constant expressions in the precondition or target involve division or remainder, the precondition must prevent compile-time division by zero.

Finally, there are some ambiguities in the IR specification, as its semantics are given informally, often by example. In some cases, such as LLVM's optional reassociation of floating-point operations, it is unclear what correctness conditions—if any—an

[3]Often, it is only practical to verify a subset of possible type assignments, as LLVM includes more than eight million integer types.

optimization must meet. Similarly, LLVM's `select` instruction can be reasonably interpreted in at least two distinct ways that are identical except in the presence of certain restricted forms of undefined behavior (see Section 3.3.6).

While developing the prototype implementation of the Alive toolkit, we extracted 334 transformations from LLVM and translated them to Alive. This represented roughly a third of the estimated number of transformations in InstCombine at the time; the remainder mostly involve instructions not included in Alive. In doing this translation, we discovered eight previously unknown bugs in LLVM, which have since been confirmed and fixed by LLVM developers. At least fifteen additional bugs have been found subsequently, and Alive has been used to prevent the introduction of incorrect optimizations [54].

We also created a version of LLVM that replaced its peephole optimization pass with code generated by the Alive toolkit that implemented the 334 transformations. This modified compiler was used to build LLVM's test suite and the SPEC INT 2000 and 2006 benchmarks. Compilation time of the benchmarks was comparable to the hand-written code. The compiled executables showed only a small performance regression compared to the full InstCombine pass.

1.4.2 Inferring Preconditions

Some optimizations are not valid for all input programs where they could apply. For example, consider a program that takes an integer and shifts its bits left c_1 places, and then left again by c_2 places. It might seem reasonable to replace these shifts with a single shift left by $c_1 + c_2$ places, but such a replacement may not preserve the meaning of the program. For example, shifting a 32-bit integer left by more than 31 places has behavior that may vary widely across processor architectures, and for this reason is explicitly undefined in LLVM IR. On Intel's X86 platform, for example, shifting a 32-bit integer left by 16 places twice will produce zero, but shifting left by 32 places leaves the input untouched. Thus, in order to be correct, our transformation must include a precondition that restricts it to applying when the shift amount is less than the bit width. Figure 1.3 shows this optimization specified in Alive.

```
Pre: C1 + C2 u< width(%x)
  %y = shl %x, C1
  %z = shl %y, C2
=>
  %z = shl %x, C1 + C2
```

Figure 1.3: Combining two shifts by constants into a single shift is only valid if the resulting shift does not overflow

Finding the best precondition for an optimization can be tricky. Developers must balance several requirements. First, the precondition must be strong enough to prevent the optimization from applying where it would in invalid. Second, the precondition should not be too strong. A weaker precondition allows the optimization to apply in more circumstances than a stronger one. Third, the precondition should not be too complex. The precondition must be evaluated for each code fragment that matches the source pattern, and a lengthy precondition may negatively affect compilation time.

To assist developers in finding preconditions, we introduce a data-driven method for precondition inference based on the Precondition Inference Engine [96]. The method uses a set of examples to learn predicates that help distinguish the cases where a transformation is correct from those where it is incorrect. These are then assembled into a precondition for that transformation.

In order to infer a precondition for a transformation, we must first find a set of examples that demonstrate its behavior. Each example represents a situation where the transformation may be applied and contains the information available to the compiler during optimization: type assignments for the values and value assignments for the symbolic constants. A *positive example* represents a situation where the transformation is correct, meaning the target refines the source when the assigned types and values are specialized into the transformation. Conversely, a *negative example* is one where the optimization is incorrect.

Note that examples are classified in terms of refinement. In particular, for an example to be positive, the transformation must be a refinement for all possible assignments of the run-time variables. For the transformation in Figure 1.3, an example will fix the types and the values of `C1` and `C2`, but we must check whether the optimization is valid

τ	C1	C2	$\pm$	C2 < C1	C1 + C2 u<= width(%x)
i8	3	2	+	$\top$	$\top$
i8	3	5	−	$\bot$	$\bot$
i32	18	15	−	$\top$	$\bot$
i64	18	15	+	$\top$	$\top$

Figure 1.4: Learning predicates for the transformation in Figure 1.3. The first three columns show four possible examples, each comprising a type (τ) and the values for `C1` and `C2`. The fourth column indicates whether the example is positive or negative. The fifth and sixth columns show two predicates that might be learned during inference, and whether they return true ($\top$) or false ($\bot$) for each example.

for all values of `%x` to classify it as positive. The first three columns in Figure 1.4 show some examples, with column four indicating whether they are positive or negative.

Examples are obtained using random selection. For a sample of possible type assignments, we choose arbitrary values for each symbolic constant and then classify the example. To ensure that we always have both positive and negative examples—and to determine whether such examples exist—we also use an SMT solver to find further examples.

Some examples represent program fragments that are unlikely to appear. For example, additions involving a constant zero are removed early on. Being able to ignore certain examples can sometimes lead to simpler preconditions. Some optimizations are valid if a condition holds, or if any symbolic constant is zero. If the optimization is only applied in situations where the constants are non-zero, then there is no advantage to weakening (and complicating) the precondition to accept those examples. We provide a method for developers to provide explicit assumptions about what examples to consider.

Using these examples, we next learn predicates. These predicates are comparisons of arithmetic expressions involving symbolic constants. For a given example, each predicate will evaluate to true or false. The goal is to find enough predicates so that every pair of positive and negative examples will be evaluated differently by at least one predicate. That is, the set of predicates are sufficient to distinguish the positive and negative examples. If this is not the case, we take a set of positive and negative examples that all have identical evaluations for the previously learned predicates and search for

a predicate that helps distinguish this set. The fifth and sixth columns of Figure 1.4 show two predicates that might be learned for the transformation in Figure 1.3 and their behavior for four examples.

The search for new predicates is based on enumeration. Using the symbolic constants and type constraints from the optimization, we generate well-typed, parametric constant expressions and predicates. We make some effort to avoid unnecessary duplication. For example, fixed-width integer addition is commutative and associative, so there is no need to generate both $a + b$ and $b + a$.

Once we have learned enough predicates, we find a formula using the predicates that accepts all the positive examples and rejects all the negative examples. We use two different Boolean formula learners in this part. The first finds increasingly-large disjunctions of predicates that accept all positive examples until it is able to reject every negative example with at least one disjunction. This is used to find *full* preconditions, and may produce very complex preconditions. The second formula learner searches for a formula that accepts as many positive examples as it can while keeping its complexity below a certain bound. This can result in far simpler *partial* preconditions at the cost of rejecting a few edge cases.

Finally, having learned a precondition, we attempt to verify the transformation using it. If the transformation and precondition are correct, we report them to the user. Otherwise, we generate a few counter-examples where the precondition holds and the transformation is incorrect, add these to the set of negative examples, and resume predicate learning. Similarly, if the precondition is intended to be full, we check for examples where the transformation is correct but the precondition does not hold, and add any we find to the set of positive examples.

This method can be used to strengthen a precondition or find a precondition for a possibly-valid transformation. It can also be used to weaken an existing precondition by finding new positive examples. (To speed up predicate learning, the predicates in the existing precondition can be used as the initial learned set.)

We used a prototype implementation to re-infer preconditions for 174 transformations derived from LLVM that required a precondition but did not require the results

of any dataflow analysis. The prototype inferred full preconditions within 1000 seconds for 133 transformations, and inferred partial preconditions for an additional 31 transformations. The prototype was additionally used to find preconditions for 71 transformations generalized from patterns found by the Souper super-optimizer [57]. The prototype inferred full preconditions for 51 transformations, and partial preconditions for an additional three.

1.4.3 Checking for Non-Termination

The primary advantage of specifying peephole optimizations in Alive is that Alive-expressed transformations are easier to reason about than programs written in a general-purpose language. We have seen this with automated verification and precondition inference, but there are further ways to analyze Alive transformations. In particular, we need not restrict ourselves to examining transformations in isolation. LLVM's InstCombine checks each instruction against a list of transformations and applies the first one that matches. Changing the order of the list may change the results of optimization, as two transformations may apply to the same input but only one will be performed.

LLVM repeatedly applies peephole optimizations until none apply, *i. e.*, the optimization reaches a fixed point. But this assumes that such a fixed point always exists. It is possible that one transformation may undo the work of another, or that a sequence of one or more transformations will result in a program where the sequence can be applied again. If the list of transformations in InstCombine is not carefully designed, some input programs may never reach a fixed point and the transformation process will run indefinitely without producing a result.

Figure 1.5 shows a pair of transformations that can lead to nontermination. The first, transformation A, moves an `and` operation inside an `xor`, possibly reducing the number of bits flipped. The second, transformation B, reduces the number of references to the value `%Y` by moving an `xor` inside an `and`. An input program matching A will result in a program matching B if `C2` and `C1 & C2` are equal. The result of applying B will be a program matching A, and the process of optimization will continue indefinitely, alternating applications of A and B.

```
Name: A                              Name: B
  %p1 = xor %W, C1                     %p2 = and %X, %Y
  %r1 = and %p1, C2                    %r2 = xor %p2, %Y
=>                                   =>
  %q1 = and %W, C2                     %q2 = xor %X, -1
  %r1 = xor %q1, C1 & C2               %r2 = and %q2, %Y
```

Figure 1.5: Two transformations that can be alternately applied to an input program indefinitely. Note that A and B are not exact inverses.

We develop a technique for detecting sequences of transformations that may lead to non-termination. First, we show how to *compose* two transformations, producing a new transformation equivalent to sequentially applying the transformations. Using this, we can produce a single transformation representing the action of a sequence of transformations. Next, we describe the conditions that must hold for an transformation to cause non-termination: (1) it must compose with itself, (2) the precondition of the self-composition must be satisfiable, and (3) the self-composition must not have a larger source pattern. The first two conditions are necessary for the transformation (or sequence of transformations) to be applied repeatedly. The third condition is necessary for this repetition to continue indefinitely.

Figure 1.6 shows transformation AB, the composition of transformations A and B, and transformation ABAB, the composition of AB with itself. The precondition for AB reflects the condition necessary for B to follow A. The precondition for ABAB, while longer, is equivalent, because the additional clause is a tautology. Because ABAB was created by matching the target of AB with the source of AB, this means that AB can be applied twice anywhere it can be applied once, and thus it can be applied arbitrarily many times.

In contrast, transformations where self-composition increases the size of the source pattern can only be applied finitely many times. For example, reassociating two addition instructions may create further opportunities to reassociate, but reassociating twice requires three addition instructions. This puts an upper limit on the number of times the reassociation transformation can be applied to a given input program.

To check whether two transformations compose, we attempt to match the target of the first transformation with the source of the second. To match, instructions must

```
Name: AB
Pre: C2 == C1 & C2
  %p1 = xor %X, C1
  %r1 = and %p2, C2
=>
  %q2 = xor %X, -1
  %r1 = and %q2, C2
```

```
Name: ABAB
Pre: C2 == C1 & C2 && C2 == -1 & C2
  %p1 = xor %X, C1
  %r1 = and %p2, C2
=>
  %q2 = xor %X, -1
  %r1 = and %q2, C2
```

Figure 1.6: Transformation AB is equivalent to applying transformations A and B from Figure 1.5 in sequence. Its precondition ensures that the result of applying A will be matched by B. Transformation ABAB is equivalent to applying AB twice. Note that it applies in exactly the same circumstances as AB.

have the same op-code, symbolic constants can match any constant (including constant expressions), and input variables may match anything. Because we are interested in cases where the second transformation is enabled by the first, we require that at least one instruction in the second's source match an instruction in the first's target. When two transformations compose, we may create a new transformation from the source of the first and the target of the second. We replace input variables with the instructions or constants they matched, if any. See Section 5.1 for further details that must be considered.

Once we implement composition, we can test whether a sequence of transformations can cause non-termination. Once we have that test, we can search for such sequences, given a set of transformations. Using 416 transformations derived from LLVM, our prototype termination checker exhaustively searched sequences of up to four transformations and sampled further sequences of up to seven transformations, finding 184 sequences that could cause non-termination. Test cases generated from these sequences were given to a version of LLVM with an Alive-generated peephole optimizer, confirming non-termination in 179 cases. The remaining five cases did not cause non-termination, due to interaction between transformations (see Section 5.4.1).

By specifying the peephole optimizations in Alive, we have enabled a high-level analysis of a set of peephole optimizations. Performing a similar analysis on LLVM's InstCombine pass, which is twenty-thousand lines of C++, is considerably more difficult.

1.5 Contributions to this Dissertation

The material in this dissertation is drawn from four previously published papers written in collaboration with my advisor Santosh Nagarakatte, Nuno Lopes, John Regehr, and Aarti Gupta. They are:

1. "Provably correct peephole optimizations with Alive" [76], which introduces Alive and describes the process for automatically verifying optimizations and generating implementations.

2. "Termination checking for peephole optimizations in LLVM" [85], which describes composition of Alive optimizations and how to determine whether an optimization can be applied indefinitely to an input program.

3. "Alive-FP: Automated verification of floating-point–based peephole optimizations in LLVM" [87], which extends Alive with reasoning about floating-point values and discusses semantics for some of LLVM's fast-math flags, which permit some optimizations by making additional assumptions.

4. "Alive-Infer: Data-driven precondition inference for peephole optimizations in LLVM" [86], which shows methods for generating preconditions for Alive optimizations that are as widely applicable as possible, or that meet a bound for precondition complexity.

1.6 Organization of this Dissertation

Chapter 2 describes LLVM and its IR in more detail and gives some background on satisfiability modulo theories. Chapter 3 discusses the Alive language and the Alive-NJ prototype, including the SMT translations used for verification, code generation, and the extension to floating-point optimizations. Chapter 4 discusses precondition inference. Chapter 5 discusses composition of Alive optimizations and the method for detecting potential non-termination.

Chapter 2

Background

Alive specifies peephole optimizations that apply to the LLVM Intermediate Representation (IR). Section 2.1 provides q quick introduction to the IR, with particular emphasis on its handling of undefined behavior. The Alive toolkit checks the correctness of a transformation with the aid of a solver for Satisfiability Modulo Theories (SMT). Section 2.2 gives an overview of SMT, with additional details about the theories of bit vector and floating-point arithmetic, which are used by Alive.

2.1 LLVM

The LLVM Compiler Infrastructure is widely used, both as a working compiler and as a subject for research. Its component technologies are designed to be modular and reusable, making it a useful basis for many external compiler-related projects.

Broadly, LLVM can be divided into three parts. A *front end* takes source code and translates it into LLVM's *internal representation* (IR). Next, LLVM applies several optimization passes, implemented as IR-to-IR translations and analyses over IR. Finally, the *back end* translates the optimized IR to assembly language or machine code for a target architecture.

LLVM provides one front end, Clang, which handles C, C++, and Objective-C. Additional front ends are provided by other projects, such as JavaScript and Swift, which can then take advantage of LLVM's optimizer and code generation stages. LLVM provides back ends for many processor architectures, such as Intel X86 and ARM. These may be used as a stand-alone compiler, or can be incorporated into larger software projects, for example, to provide just-in-time compilation of scripts in a web browser.

2.1.1 The LLVM Intermediate Representation

The LLVM Intermediate Representation (IR) is a common interface between LLVM's various front-ends, back-ends, and "middle-end" optimization passes. Unusually, the LLVM IR is itself a low-level programming language with concrete syntax and an independently-specified semantics [2]. This modularity makes it easier for other projects to reuse parts of LLVM: compiler front-ends for new languages can target LLVM IR and take advantage of LLVM's low-level optimizations and architecture-specific back-ends. Consumers of LLVM IR include LLVM's existing back-ends, an interpreter, and a just-in-time compiler.

The IR resembles a typed assembly language. A program contains one or more functions, each of which has formal parameters and one or more basic blocks containing instructions. Each block has a single entry point and concludes with a conditional or unconditional branch instruction. Instructions have one or more arguments, and may have an additional return value. The return value of an instruction is stored in a unique temporary register variable and may be used as an argument to later instructions. LLVM IR programs are expressed in static single-assignment (SSA) form: each variable is defined exactly once and its value is never reassigned. In order to express control structures such as loops, LLVM IR includes special instructions called ϕ nodes. These always occur at the start of a black and assign a value to a variable based on how control reached the block.

Figure 2.1 shows a function defined in C and its translation to LLVM IR. Noteworthy features include the branch instruction, `br`, which takes a destination label for unconditional branches, and a Boolean value and two destination labels for conditional branches. Block `test` contains two `phi` instructions, which select values for `%z1` and `%x1` based on whether control reached `test` from `entry` or `body`.

The IR is strongly typed. Each value (instruction result, formal parameter, or constant) has a type, and each instruction has rules that determine whether its arguments are properly typed. Many IR instructions are type parametric and can be used with different types, as long as typing rules are satisfied. For example, the `add` instruction

```
int factorial(int x) {
  int z = 1;
  while (x > 1) {
    z = z * x;
    x = x - 1;
  }
  return z;
}
```

(a)

```
define i32 @factorial(i32 %x0) {
entry:
  br label %test

test:
  %z1 = phi i32 [1, %entry], [%z2, %body]
  %x1 = phi i32 [%x0, %entry], [%x2, %body]
  %b  = icmp sgt i32 %x1, 1
  br i1 %b, label %body, label %exit

body:
  %z2 = mul nsw i32 %z1, %x1
  %x2 = sub nsw i32 %x1, 1
  br label %test

exit:
  ret i32 %z1
}
```

(b)

Figure 2.1: A function to compute factorials written in C and compiled to LLVM IR

can accept arguments of arbitrary integer types, as long as both arguments and the result have the same type. LLVM includes many integer types, which are bit vectors of a specific width. Figure 2.1 includes both 32- and 1-bit integers, the types for which are written `i32` and `i1` respectively. The maximum bit width for an integer is $2^{23} - 1$. LLVM provides six floating-point types. Pointer types in LLVM specify the type of the pointer target, *e. g.*, `i32*` is a pointer to a 32-bit integer. There are also multiple aggregate types, including constant-length arrays of arbitrary values, structures, and vectors. Many instructions that are defined for integers or floating-point values also work with integer or floating-point vectors.

2.1.2 Undefined behavior

The LLVM IR is designed to enable efficient compilation of C and C++ programs to a processor's instruction set. Both the input and output languages leave certain operations undefined. Sometimes, operations are left undefined because they cannot be defined, such as division by zero. Other times, leaving an operation undefined gives more freedom to the implementation of a processor or compiler. Leaving behavior undefined for certain inputs simplifies the architecture of a processor by eliminating the

need for checking the inputs: it is the program's responsibility to avoid those inputs, and no behavior by the processor in response can be considered incorrect. For a compiler, undefined behavior can enable optimizations that change a program's behavior only in those cases where it has no defined behavior. For example, signed addition overflow is undefined in C, so a compiler is free to rewrite $a+b > a$ to $b > 0$, because the only cases where those expressions are not equal are ones where $a + b$ overflows. The LLVM IR also leaves some operations undefined, in order to accurately capture the semantics of C and C++, to take advantage of optimization opportunities, and to simplify the translation to machine instructions.

For example, the behavior of shift operations when the shift amount exceeds the bit width of its result varies wildly between processor architectures. If LLVM mandated a particular behavior—even something seemingly reasonable, such as returning zero—it would need additional instructions on architectures with different behavior in order to preserve its semantics. This would be especially unfortunate when compiling C programs, as C also does not define semantics for shifts when the shift amount is too large. By over-defining shifts, LLVM would require itself to produce slower programs on some architectures in order to control its behavior under circumstances that, in a correctly written program, should never occur anyway.

The presence of undefined behavior in LLVM IR does not prevent it from safely compiling languages where behavior is more defined. Instead, the IR allows splitting a safe operation, such as an array access with bounds-checking, into two parts: the bounds check and the (potentially undefined) memory access. If a function repeatedly accesses a specific array element, LLVM can optimize by consolidating the bounds checks and leaving the individual memory accesses unguarded.

Deferred undefined behavior

Although undefined behavior gives a compiler freedom when transforming code, compilers must be careful to avoid introducing undefined behavior. If a transformation results in a program that is not defined for some input where the original program was defined, then there is no guarantee that the transformed program will have the same behavior

```
for (i = 0; i < N; i++) {
    a[i] = a[i] & (1 << B);
}
```

(a)

```
int m = 1 << B;
for (i = 0; i < N; i++) {
    a[i] = a[i] & m;
}
```

(b)

Figure 2.2: Two C programs that mask bits from an array of integers. Unlike (a), (b) is undefined when `N` is zero and `B` is larger than the width of an integer.

as the original for that input. The possibility of introducing undefined behavior can prevent certain optimizations, such as speculative execution.

Consider a loop that contains a loop-invariant computation that might be undefined for some inputs, such as the shift in Figure 2.2(a). Moving this common computation outside the loop, as in Figure 2.2(b), avoids duplicating work, but introduces undefined behavior in the case where the computation is undefined but the loop does not execute: the original program does nothing, but the optimized program has undefined behavior.

To allow for speculative execution and similar optimizations, LLVM distinguishes between undefined behavior that occurs immediately and undefined behavior that is delayed until its result is used. If the result of a computation with deferred undefined behavior is not used in an externally visible way, the program as a whole is still defined. If shifting has deferred undefined behavior when the shift amount is too large, then the programs in Figure 2.2 have the same behavior for all values of `N` and `B`.

Undefined values

What is the behavior of an instruction that has deferred undefined behavior? One possibility in LLVM is to return an *undefined value*, or `undef`. This represents a non-deterministic bit pattern or, equivalently, a set of possible values. LLVM uses `undef` to represent uninitialized variables, the initial contents of memory, padding bits in structures, and the results of certain instructions, such as shifts where the shift amount is too large (but see Section 3.2.5).

A fully-undefined value may be any value of its type, but subsequent operations may partially define it. If a 32-bit value `%a` is undefined, then `shl i32 %a, 1` will result

in a value where the low bit is 0 and all higher bits are undefined. When interpreting undefined values as sets of possible values, `%a` is a set A containing all 32-bit values, and the result of the shift will be $\{a \times 2^1 : a \in A\}$, *i. e.*, the set of all even 32-bit integers.

Notably, undefined values are not required to make consistent non-deterministic choices. That is, `xor i32 %a, %a` may take any value if `%a` is undefined. In terms of sets, its possible values are $\{a_1 \oplus a_2 : a_1 \in A, a_2 \in A\}$ (where $\oplus$ is exclusive or). This can be surprising, as it violates the interpretation of LLVM IR as machine instructions manipulating concrete values, but this frees LLVM from having to remember the specific arbitrary bit pattern chosen for an undefined value. Additionally, optimizations are free to choose convenient replacements for undefined values without needing to consider all uses of the undefined value.

This freedom does make other transformations problematic. A transformation that increases the number of uses for a value may introduce new behavior when that value is undefined. For example, replacing $a \times 2$ with $a + a$ introduces odd numbers when a is an undefined value.

Poison values

Another way to represent deferred undefined behavior is specially mark the results of an operation that has or relies on deferred undefined behavior. If a marked value is used with an operation that has visible side-effects, such as a write to volatile memory or a conditional branch, the operation has undefined behavior. In LLVM, this marked value is called a *poison value*.

Poison values were introduced to model signed arithmetic in C. Initially, this was modeled by adding `nsw` ("no signed wrap") attributes to certain arithmetic instructions, which indicated that the instruction returned an undefined result if signed overflow would have occurred. This turned out to be insufficient to allow some optimizations, such as widening [41]. Because signed arithmetic overflow is undefined, it is possible to perform operations in a larger type that is efficient for the processor without changing

their semantics. For example, `int` in C has 32 bits,[1] but on a 64-bit architecture it may be more convenient to treat `int` values as 64-bit `long` values (this avoids the need for sign extension when calculating an array offset, for example).

Consider two `int` variables, `a` and `b`, and the expression `(long)(a + b)`, which adds them and then extends the value to a 64-bit type. This would be translated to:

```
%0 = add nsw i32 %a, %b
%1 = sext i32 %0 to i64
```

After widening, `%a` and `%b` will become `i64` values and the `sext` (sign-extend) instruction is dropped. But can this transformation introduce new behavior? Before widening, the upper 33 bits of `%1` will be all ones or all zeros, depending on the sign of `a + b`. Even if the addition overflows, an undefined 32-bit value sign-extended to 64 bits will have this property. After widening, an addition that previously overflowed will easily fit into 64 bits, resulting in no overflow but producing values that are not the result of sign-extending any 32-bit value. If `a` is 1 and `b` is $2^{31} - 1$, their sum after widening will be 2^{31}, which is not representable as a signed 32-bit integer.

Instead, `add nsw` returns a poison value when signed overflow occurs. Unlike undefined values, poison values never become partially defined: an instruction with a poison argument will always return poison or have immediate undefined behavior (but see Section 3.3.6). Thus, sign-extending a poison value results in a larger poison value, which the compiler may replace with any value in the larger type, including those that cannot be produced by sign-extending an arbitrary value in the smaller type.

Unfortunately, LLVM has not always been consistent with how optimizations interpret poison values. For example, certain optimizations are valid only if conditional branches have undefined behavior when the condition is poison, while others require them to select one of the destinations non-deterministically. Recent work [65] has proposed a more consistent semantics, and combined the ideas of `undef` and poison values, but those changes have not yet been adopted.

---

[1] Formally, `int` may have any size no smaller than 16 bits, but C compilers on many 64-bit architectures use 32 bits to save space and remain compatible with 32-bit code.

```
if (c != 0) {
    erase_hard_drive();
} else {
    greet_friends();
    d = x/c;
}
```

(a)

```
  %b = icmp ne i32 %c, 0
  br i1 %b, label %then, label %else

then:
  call void @erase_hard_drive()
  br label %after

else:
  call void @greet_friends()
  %d = sdiv i32 %x, %c
  br label %after
```

(b)

Figure 2.3: A program fragment that has undefined behavior if `c` is zero. The compiler can conclude that `else` is only executed when `%c` is zero, so the conditional branch can be replaced with an unconditional branch to `then`.

Immediate undefined behavior

Not all undefined behavior can be deferred. Division by zero, for example, will typically cause the processor to interrupt the current process and invoke a trap handler. Speculatively executing division operations will therefore change the behavior of a program unless the compiler can show that the divisor is always non-zero.

In general, an operation that has undefined behavior may be replaced with the `unreachable` instruction, which indicates to the compiler that control flow will never enter a particular basic block. This means that an operation with undefined behavior can change the behavior of previous operations. Consider the program in Figure 2.3, which erases your hard drive if `c` is zero, and otherwise greets your friends and then divides by `c`. The compiler can show that `c` is always zero in the else block (label `else` in the IR code), and thus the computation of `d` always has undefined behavior, meaning that the entire else block is unreachable. The conditional branch can be replaced with an unconditional branch to `true`. The program instead always erases your hard drive and never greets your friends. In effect, the division by zero has not only erased your hard drive, it has cancelled the preceding function call to `greet_friends`.

In C, if a program execution includes an operation with undefined behavior, the behavior of the entire execution is undefined: there is no guarantee that any event will or will not happen before or after the undefined operation. It is unclear whether the

LLVM IR has similar freedom, but it is clear that (1) any behavior after an operation with undefined behavior is also undefined, and (2) the compiler is free to assume that control never reaches a block that has undefined behavior.

2.2 Satisfiability Modulo Theories

Satisfiability, or SAT, problems are Boolean formulae containing free Boolean variables and logical connectives. For example, $\neg p \vee (q \wedge \neg p)$ includes the Boolean-valued variables p and q. If there is an assignment of truth values to the variables such that the formula evaluates to true (*i. e.*, is *satisfied*), the problem is *satisfiable*. Otherwise, it is *unsatisfiable*.

In general, solving a SAT problem is NP-complete, with the best known algorithms requiring time exponential in the size of the formula in the worst case. Fortunately, many applications of SAT solving have been found that do not require solving these worst-case formulae, and SAT solving is now considered tractable for many purposes.

Satisfiability modulo theories (SMT) extends the scope of SAT solving by introducing non-Boolean variables and expressions, which are defined by a *theory*. For example, the theory of quantifier-free linear integer arithmetic (QF-LIA) introduces integer variables, integer-valued functions such as addition and multiplication by an integer constant, and Boolean-valued functions such as equality testing of integer-valued expressions.

As with SAT problems, algorithms exist to efficiently solve some SMT problems. There is a standard format for specifying SMT problems, SMT-LIB, and multiple SMT solvers exist that focus on solving SMT problems using a particular theory, or use heuristics to choose strategies for solving general SMT problems. Figure 2.4 shows an SMT problem specified in the SMT-LIB format, which declares an integer variable x and asserts that $x \cdot x < 0$. The assertion is not true for any integer, so an SMT solver will report that the problem is unsatisfiable.

SMT solvers check whether any valuation of variables will satisfy a formula. That is, the formula implicitly has an existential quantifier. To prove that a property holds

```
(declare-fun x () Int)
(assert (< (* x x) 0))
(check-sat)
```

Figure 2.4: An SMT problem written in SMT-LIB format that is satisfiable if and only if $\exists_{x \in \mathbb{Z}} : x^2 < 0$

for all values, for example $\forall_{x \in \mathbb{Z}} : x^2 \geq 0$, an SMT solver can be asked for a valuation where the property does not hold, such as $\exists_{x \in \mathbb{Z}} : x^2 < 0$. If that negated problem is unsatisfiable, this proves the correctness of the original statement.

In many cases, an SMT solver given a satisfiable problem can show that the problem is satisfiable by providing a concrete valuation of the variables such that the formula evaluates to true. This valuation is called a model. In this context, the problem can be considered a query that requests the solver to find a model satisfying a condition. When the query is the negation of a desired universal property, any models represent counter-examples demonstrating that the property does not hold.

The Alive toolkit discussed in this dissertation uses the Z3 SMT solver [28] to check the correctness of transformations of LLVM IR code. To represent the semantics of integer and floating-point operations, it encodes them into SMT using the theories of quantified bit vectors and floating-point arithmetic.

2.2.1 SMT bit vector theory

The quantified and quantifier-free bit vector theories of SMT (BV and QF-BV, respectively), describe modular integer arithmetic and bit-wise logical operations that can be used to model fixed-width machine integers. A bit vector is simply a list of n Boolean values, where n is fixed by the vector's *sort* (*i. e.*, type). Functions that accept bit vector arguments are often parametric over sorts. For example, the arguments and return value of addition (`bvadd`) may be bit vectors of any width, so long as all three have the same width.

Bit vectors may be interpreted as unsigned integers or as signed integers in 2's-complement format. For several operations, such as addition, subtraction, and multiplication, these interpretations coincide and a single function is used for both. Other

operations, such as division, have separate signed and unsigned variants. Inequality tests similarly provide signed an unsigned variants.

A bit vector with width w represents unsigned values between 0 and $2^w - 1$, inclusive, and signed values between -2^{w-1} and $2^{w-1} - 1$, inclusive.

2.2.2 SMT floating-point arithmetic theory

The floating-point theory in SMT (SMT-FPA) [17] models arithmetic over fractional values with semantics similar to those specified by IEEE 754 [3]. The sort of floating point numbers $\mathbb{F}_{E,M}$ is parameterized by two positive integers, the exponent width E and the significand width M. The floating-point sorts include the (binary) floating-point types defined by IEEE 754, as well as other types with similar behavior.

A value of $\mathbb{F}_{E,M}$ is either NaN or a triple $\langle s, e, m \rangle$ containing three bit vectors: the sign bit s has 1 bit; the exponent e has E bits, and the mantissa m has $M - 1$ bits. The missing bit in m is the implicit integer part, which is always 0 or 1 depending on the exponent. Non-NaN values in $\mathbb{F}_{E,M}$ can be mapped to the extended real numbers $\mathbb{R} \cup \{+\infty, -\infty\}$ using the function $v_{E,M}$:

$$v_{E,M}(\langle s, e, m \rangle) = \begin{cases} (-1)^s \cdot 2^{1-bias(E)} \cdot (0 + \frac{m}{2^{M-1}}) & \text{if } e = 0 \\ (-1)^s \cdot 2^{e-bias(E)} \cdot (1 + \frac{m}{2^{M-1}}) & \text{if } 0 < e < 2^E - 1 \\ (-1)^s \cdot \infty & \text{if } e = 2^E - 1 \wedge m = 0 \end{cases} \tag{2.1}$$

where $bias(E) = 2^{E-1} - 1$, and the bit vectors are interpreted as unsigned integers.

Arithmetic over values in $\mathbb{F}_{E,M}$ returns the exact value that would be computed in the extended reals, if that value is in $\mathbb{F}_{E,M}$. Otherwise, it will return one of the two values in $\mathbb{F}_{E,M}$ that are immediately greater or smaller. The specific choice is made according to the *rounding mode*, which is one of: round towards zero, round towards positive, round towards negative, round to nearest (ties toward zero), and round to nearest (ties toward even). All arithmetic operations in SMT-FPA have an explicit rounding mode parameter, except for remainder.

Certain operations that are not defined in real arithmetic will return infinite values in floating-point, such as dividing a non-zero value by zero. Similarly, operations that

return results too large to be represented as finite values may also produce infinite values.

Finally, NaN represents an error condition or ill-defined expression, such as $\infty - \infty$ or $0 \div 0$. All floating-point arithmetic operations return NaN if any of their arguments are NaN. All floating-point comparisons are false if one or more argument is NaN. That is, NaN is not greater than, less than, or equal to any floating-point value including itself.

SMT-FPA models IEEE 754 at the floating-point data specification level, which has a single NaN value, rather than the representation or bit-string specification levels, which have multiple NaN values. This prevents SMT-FPA from meaningfully reasoning about certain operations, such as copying the sign of a NaN value or the total ordering predicate.

Chapter 3

Specifying Peephole Optimizations with Alive

Peephole optimizations rewrite small portions of intermediate representation (IR) code in order to improve efficiency or to present code in a canonical form expected by other optimization passes. To be considered correct, this rewrite must not change the behavior of the program as a whole. It can be difficult to ensure that a peephole optimization is correct, due to the complexity of the IR semantics—especially when that IR includes undefined behavior.

Alive[1] is a domain-specific language for specifying peephole optimizations over the LLVM IR intended to simplify the creation of correct transformations. Transformations specified in Alive can be automatically checked for correctness, even in the presence of one or more of LLVM's forms of undefined behavior. When the Alive toolkit determines that a transformation is incorrect, it returns an example to the user showing how the transformation changed the behavior of the program. To ensure that the implementation of the transformation is kept in sync with its correctness-checked specification, the toolkit can translate the transformation into a C++ implementation. Figure 3.1 shows the process of verifying and generating code for a transformation specified in Alive.

Transformations in Alive have three parts. The *source* and *target* describe an IR code fragment, or a set of code fragments that have a similar structure. These fragments include free variables, which represent constants or the results of other operations that are not part of the pattern. The transformation indicates that code matching the source should be replaced by the target. The *precondition* specifies additional requirements for the transformation. A code fragment that matches the source but does not satisfy the precondition will not be replaced.

---

[1] **A**utomatic **L**LVM's **I**nstcombine **Ve**rifier.

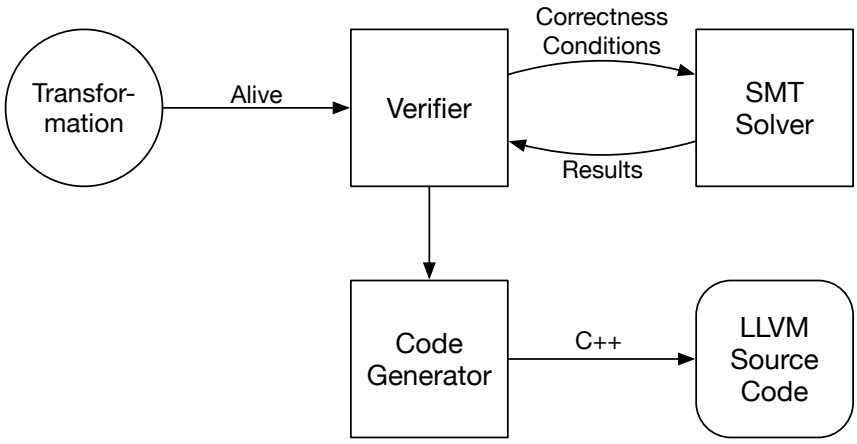


Figure 3.1: Alive transformations are automatically checked for correctness and can be translated into C++ implementations for use in LLVM

A transformation is considered correct if performing the transformation does not change the behavior of a program. Because some operations in LLVM have undefined behavior, the source and target are not always deterministic. Thus, a more precise requirement is that the behaviors of the source for a particular valuation of inputs include every possible behavior of the target, or

$$\forall_{\mathcal{I}} : B(S) \supseteq B(T), \tag{3.1}$$

where $B(S)$ and $B(T)$ represent the sets of possible behaviors for the source and target, respectively, for a valuation of the variables in $\mathcal{I}$. That is, the target *refines* the source for all input values.

For example, a transformation that rewrites $0 \div x$ to 0 is correct, even though the source has undefined behavior when $x = 0$. Because $0 \div 0$ is undefined, its set of possible behaviors includes all behaviors. Rewriting it to 0 refines the set of possible behaviors when $x = 0$ to a single behavior.

Alive is designed so that transformations can automatically be checked for correctness, without any user intervention. Correctness checking is performed by encoding the correctness conditions for the transformation into logical statements and using an SMT solver to check for counter-examples, where the correctness conditions do not hold. Sections 3.2 and 3.3 describe this process in more detail.

Additionally, Alive is designed so that transformations can automatically be translated into C++ implementations suitable for use in LLVM. This ensures that the implementation of the transformation matches its specification. Section 3.4 describes the specifics of this translation.

This chapter primarily describes the dialect of Alive accepted by the Alive-NJ prototype. This includes the floating-point extensions that were introduced by Alive-FP [87], but does not include some experimental features mentioned in the original Alive paper [76]. These features are briefly discussed in Section 3.5. Alive-NJ itself is described in more detail in Section 3.6.

Alive has been adopted by LLVM developers, who use it to check the correctness of proposed additions to the peephole optimizer. Additionally, Alive has been used to find bugs in existing LLVM code. While developing Alive, we extracted 334 transformations from InstCombine and expressed them in Alive. Doing so uncovered eight previously unknown bugs. Subsequent work has found fifteen additional bugs. Section 3.7 provides additional information about bugs found with Alive and the performance of its verification engine and of its generated code.

3.1 The Alive Language

An Alive transformation describes a process for recognizing an instruction and replacing it with a new instruction. The task of recognizing an instruction is broken into two parts. The *source pattern* describes an instruction, known as the *root*, by giving its opcode and describing its arguments, which may be other instructions or variables. The *precondition* describes conditions that must be satisfied by the variables and instructions mentioned in the source pattern. The *target pattern* describes the instruction or value that will replace the root, and may also describe new instructions that must be added to the program in order to create the new root.

Figure 3.2 shows two transformations. Each has the structure $source \implies target$, optionally preceded by a precondition. The source patterns are given as a list of declarations assigning register names to instructions. The final instruction in the source

```
Pre: C1 == C2 + 1 && \
    countTrailingZeros(C1) == 0
  %Y   = and %Z, C2
  %LHS = xor %Y, C1
  %r   = add %LHS, %RHS
=>
  %or  = or %Z, ~C2
  %r   = sub %RHS, %or
```

(a)

```
  %c = icmp ult %x, 0
  %r = select %c, %A, %B
=>
  %r = %B
```

(b)

Figure 3.2: Two transformations demonstrating several features of Alive

pattern is the root, named `%r` in these examples. The other instructions describe the values mentioned in the arguments to the root. In Figure 3.2(a), the arguments to the root are `%LHS` and `%RHS`. The former is declared to be an `xor` operation whose arguments are `%Y` and a symbolic constant `C1`. *Symbolic constants* are values that will be constant in the program being compiled.[2] This means their values will be known during compilation, in contrast to register variables such as `%RHS`, which may not be known until the code is executed.

The target pattern, located after the arrow, is also given as a list of declarations. The final declaration uses the same name as the source root and indicates the value that will replace it. This may be a new instruction, as in Figure 3.2(a), or some other value such as a constant or a register declared by the source, as in Figure 3.2(b). The target may also define additional instructions, which are used by the target root. These will be added to the program before the source root is replaced by the target root.

If present, the precondition is located before the source pattern and indicated by the prefix `Pre:`. It is a Boolean-valued formula using the logical connectives `&&` (and), `||` (or), and `!` (not), as well as comparisons and named predicate functions. The precondition is considered to be satisfied if it evaluates to true.

Note that the application of an Alive transformation only rewrites the root instruction, and potentially adds additional instructions. The other instructions referenced in the source must be left in place, in case their results are used by other parts of the

[2]LLVM's constant expressions also include values that will not be known until link-time (*i. e.*, after optimization), but Alive does not represent these as symbolic constants.

$$
\begin{array}{rcl}
\mathit{trans} & ::= & \mathit{pre}\ \overline{\mathit{stmt}} \implies \overline{\mathit{stmt}} \\
\mathit{stmt} & ::= & \mathit{reg} = \mathit{inst} \mid \mathit{reg} = \mathit{val} \mid \mathit{sym} = \mathit{const} \\
\mathit{val} & ::= & \mathit{reg} \mid \mathit{const} \mid \textbf{undef} \mid \textbf{poison} \\
\mathit{inst} & ::= & \mathit{binop}\ \overline{\mathit{attr}}\ \mathit{type}\ \mathit{val}, \mathit{val} \mid \mathit{fbinop}\ \overline{\mathit{fattr}}\ \mathit{type}\ \mathit{val}, \mathit{val} \\
 & & \mathit{conv}\ \mathit{type}\ \mathit{val}\ \textbf{to}\ \mathit{type} \mid \textbf{select}\ \mathit{val}, \mathit{type}\ \mathit{val}, \mathit{type}\ \mathit{val} \mid \\
 & & \textbf{icmp}\ \mathit{rel}\ \mathit{type}\ \mathit{val}, \mathit{val} \mid \textbf{fcmp}\ \overline{\mathit{fattr}}\ \mathit{frel}\ \mathit{type}\ \mathit{val}, \mathit{val} \\
\mathit{type} & ::= & \star \mid \textbf{int}\ w \mid \textbf{half} \mid \textbf{float} \mid \textbf{double} \mid \textbf{fp128} \mid \textbf{x86\_fp80} \\
\mathit{binop} & ::= & \mathit{bop} \mid \mathit{ibop} \\
\mathit{bop} & ::= & \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{sdiv} \mid \textbf{srem} \mid \\
\mathit{ibop} & ::= & \textbf{udiv} \mid \textbf{urem} \mid \textbf{shl} \mid \textbf{ashr} \mid \textbf{lshr} \mid \textbf{and} \mid \textbf{or} \mid \textbf{xor} \\
\mathit{attr} & ::= & \textbf{nsw} \mid \textbf{nuw} \mid \textbf{exact} \\
\mathit{fbinop} & ::= & \textbf{fadd} \mid \textbf{fsub} \mid \textbf{fmul} \mid \textbf{fdiv} \mid \textbf{frem} \\
\mathit{fattr} & ::= & \textbf{ninf} \mid \textbf{nnan} \mid \textbf{nsz} \\
\mathit{conv} & ::= & \textbf{zext} \mid \textbf{sext} \mid \textbf{trunc} \mid \textbf{ZextOrTrunc} \mid \textbf{fpext} \mid \textbf{fptrunc} \mid \\
 & & \textbf{fptosi} \mid \textbf{fptoui} \mid \textbf{sitofp} \mid \textbf{uitofp} \mid \textbf{bitcast} \\
\mathit{rel} & ::= & \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{frel} & ::= & \textbf{oeq} \mid \textbf{one} \mid \textbf{oge} \mid \textbf{ogt} \mid \textbf{ole} \mid \textbf{olt} \mid \\
 & & \textbf{ueq} \mid \textbf{one} \mid \textbf{uge} \mid \textbf{ugt} \mid \textbf{ule} \mid \textbf{ult} \\
\mathit{const} & ::= & \mathit{sym} \mid \mathit{lit} \mid \mathit{flit} \mid \mathit{uop}\ \mathit{const} \mid \mathit{const}\ \mathit{binop}\ \mathit{const} \mid \mathit{cfun}(\overline{\mathit{val}}) \\
\mathit{uop} & ::= & \textbf{not} \mid \textbf{neg} \\
\mathit{pre} & ::= & \textbf{true} \mid \mathit{const}\ \mathit{rel}\ \mathit{const} \mid \mathit{pfun}(\overline{\mathit{val}}) \mid \\
 & & !\,\mathit{pre} \mid \mathit{pre}\ \&\&\ \mathit{pre} \mid \mathit{pre}\ ||\ \mathit{pre}
\end{array}
$$

Figure 3.3: Abstract syntax of Alive. Arguments to *cfun* and *pfun* must follow the syntax given in Table 3.1. Lexical syntax for the metavariables *reg* and *sym* and the literals *lit* and *flit* is given in Table 3.2. The production w may be any positive integer.

program. If, after replacing the root, these results are no longer used, the instructions will be removed by a separate dead code elimination pass.

3.1.1 Abstract syntax

Figure 3.3 shows the abstract syntax of Alive. A transformation (*trans*) is a precondition (*pre*), a list of statements ($\overline{\mathit{stmt}}$) representing the source pattern, and a list of statements representing the target pattern. Statements assign a value to a variable, either a register (*reg*) or a symbolic constant (*sym*).

Register names have two purposes in Alive. Registers that occur in the left side of a statement are used to refer to an instruction or other value in subsequent instructions. Registers that only occur on the right sides of statements are *input variables*, which represent arbitrary values that might be the results of unspecified instructions, constants, or function parameters.

Symbolic constants and literal constants (*lit*, *flit*) represent in-line constants that will be present in the program being transformed. Literal constants (*e. g.*, `0` and `1`) and floating-point only literals (*e. g.*, `0.0` and `Inf`) represent exactly those constants. Symbolic constants represent constants whose values will not be known until compilation.

The remainder of the grammar can be divided into four parts. The precondition language (*pre*), which represents Boolean predicates that are evaluated at compile-time, the constant language (*const*), which represents compile-time computations that produce LLVM values, the instruction language (*inst*), which represents computation performed by the program being transformed, and the type language (*type*), which classifies the results computed by the constant and instruction languages.

Both the precondition and constant languages include a set of built-in functions. Each function has a fixed number of parameters, each of which may be a constant expression or an arbitrary value. Table 3.1 lists all the built-in functions according to the forms of argument they accept. Functions that have non-constant parameters may rely on the syntactic structure of a program (*e. g.*, `hasOneUse`) or on a dataflow analysis (*e. g.*, `isPowerOf2`).

Variable scope

Alive variables may be classified in two ways. First, they may belong to one of two grammatical forms, *reg* (register variable) or *sym* (symbolic constant). Second, they may be *bound*, meaning they occur to the left of an equal sign in a statement, or *free*, meaning they do not. Variables in Alive are not pre-declared. Instead, they are defined by the first statement that uses them.

All free variables must be defined by the source. When applying a transformation, these variables will be matched with values in the program being transformed. The target and precondition cannot define new free variables, as they do not perform matching.

Variables are introduced by statements. The variable to the left of the equals sign is bound, while any new variables on the right side will become free variables. Variables introduced in a statement are in scope for any subsequent statements.

cfun	*pfun*
(*const*)	
abs	**isShiftedMask**
countLeadingZeros	**isSignBit**
countTrailingZeros	**fpInteger**
fpext	
fptosi	
fptoui	
fptrunc	
log2	
trunc	
sext	
sitofp	
uitofp	
zext	
(*const*, *const*)	
max	**fpIdentical**
min	**WillNotOverflowSignedMul**
umax	**WillNotOverflowUnsignedMul**
umin	**WillNotOverflowUnsignedShl**
(*val*)	
computeKnownOneBits	**CannotBeNegativeZero**
computeKnownZeroBits	**isConstant**
ComputeNumSignBits	**isPowerOf2**
fpMantissaWidth	**isPowerOf2OrZero**
width	
(*val*, *val*)	
	WillNotOverflowSignedAdd
	WillNotOverflowSignedSub
	WillNotOverflowUnsignedAdd
	WillNotOverflowUnsignedSub
(*val*, *const*)	
	MaskedValueIsZero
(*reg*)	
	hasNoInf*
	hasNoNaN*
	hasNSW*
	hasNSZ*
	hasNUW*
	hasOneUse
	isExact*

Table 3.1: The constant and predicate functions (*cfun* and *pfun*, respectively), grouped according to syntax of their parameters. For example, **abs** takes a single parameter that must be a *const*.
* The argument must be bound to an instruction that can have this attribute.

```
Pre: fpext(C2) == C1
  %x = fpext %a
  %r = fcmp ole %x, C1
=>
  C2 = fptrunc(C1)
  %r = fcmp ole %a, C2
```

Figure 3.4: A transformation including a bound symbolic constant (`C2`), which is used by the precondition

Alive represents fragments of code in static, single-assignment (SSA) form, so no variable may be defined more than once. Therefore, it is an error if, say, a register is used as an argument in one statement (defining it as an input variable) and later used in the left side of a statement (re-defining it as a bound register variable). This requirement also prevents circular references, as an attempt to refer to an instruction before the statement it occurs in will be interpreted as an attempt to re-define an input variable and rejected.

All variables defined in the source are in-scope in the precondition. All variables excluding the source root are in-scope in the target. The final statement in the target must define a bound variable with the same name as the source root.

All bound variables in the source, excluding the root, must be used by at least one subsequent instruction in the source. Similarly, all bound variables in the target, excluding the root, must be used by at least one subsequent instruction in the source.

Bound symbolic constants may only be defined by the target, but are made available in the precondition. Because they represent computation performed by the compiler using information derived from the source pattern, it is feasible to use these values in the precondition, as though the variable had been replaced by its definition. However, not all a bound symbolic constants can be replaced by their definitions. For example, in Figure 3.4, expanding the definition of `C2` in `fpext(C2)` yields `fpext(fptrunc(C1))`, which is ambiguously typed and would make the transformation type incorrect (see Section 3.1.3).

Contextual restrictions

To simplify the grammar, certain restrictions are not explicitly stated. The lists of statements in the source and target must be non-empty. The attribute lists ($\overline{attr}$ and $\overline{fattr}$) should not contain duplicates. The attributes for the integer binary operator instructions must be appropriate to the opcode. The **nsw** and **nuw** attributes may occur with the **add**, **sub**, **mul**, and **shl** opcodes. The **exact** attribute may occur with the **sdiv**, **udiv**, **srem**, **urem**, **ashr**, and **lshr** opcodes.

More significantly, constant expressions in the source may only include symbolic constants and literals. This restriction avoids ambiguous variable definitions. For example, an expression such as `C1 + C2` that occured in the source would match an in-line constant in an LLVM IR program, but would not uniquely determine the values of `C1` and `C2`, making the behavior of the precondition and target of the transformation implementation dependent. If `C1` and `C2` are also constrained by other parts of the source, then the expression can be rewritten to use a new variable `C3` and a new predicate `C3 == C1 + C2` can be added to the precondition.

3.1.2 Concrete syntax

The grammar of written Alive is very similar to the abstract syntax given in Figure 3.3, with a few changes and additions.

An Alive transformation may optionally begin with a name declaration. This is indicated with the prefix `Name:`. The rest of the line (excluding any whitespace immediately following the prefix) is considered the name. The contents of the name have no direct impact on a transformation, but may be used when reporting information to the user. Names must be used when giving multiple transformations in a single file, in order to make it clear where a transformation begins.

Whitespace characters, such as spaces and tabs but excluding newlines, may delimit grammatical tokens, but are otherwise not significant. Alive is line-oriented, so the precondition and each statement will be terminated by a newline. Lines that are too long can be continued by ending the line with a backslash (\).

Grammar production	Lexical form
reg	`%[0-9A-Za-z_.]+`
sym	`C[0-9]*`
lit	`-?[0-9]+`
flit	`-?[0-9]+.[0-9]+ | -?inf | nan`
int *width*	`i[0-9]+`

Table 3.2: Concrete syntax for selected Alive productions. The production *width* may be any positive integer.

Comments are introduced by a semicolon (`;`) and continue to the end of the line.

The source and target of a transformation both contain a list of statements. These statements are presented on separate lines, with the `=>` separator also on its own line. The lists of attributes are separated by whitespace. The lists of arguments to constant and predicate functions are separated by commas.

The variable productions *reg* and *sym*, literal productions *lit* and *flit*, and type form **int** *width* must conform to the lexical rules given in Table 3.2. For *flit*, the strings `nan`, `inf`, and `-inf`, represent Not A Number, infinity, and negative infinity, respectively. For **int**, the digits following the `i` indicate the *width*.[3]

The *type* production is always optional. When the type is omitted, its production is $\star$. For example `add %x, %y` becomes **add** $\star\, x, y$. Conversion instructions include a `to` keyword that is only used if the second (destination) type is explicitly given. Thus, **zext int** $16\, x$ **to** $\star$ is written `zext i16 %x`.

For brevity, the abstract grammar re-uses the *rel* production to represent comparison predicates, and re-uses *binop* opcodes to represent binary constant expressions. The concrete syntax uses operators for these. Table 3.3 maps the productions in the abstract grammar to operators in the concrete grammar.[4] Thus, for example, the production C_1 **ult** C_2 would be written `C1 u< C2`.

[3]The syntax allows a zero-width `i0`, but this will be rejected by the type checker.

[4]The concrete syntax for **urem** used in a constant expression, `%u`, is also a valid variable name. Fortunately, it is always clear from context which is intended.

eq	`==`	**ne**	`!=`	**add**	`+`	**udiv**	`/u`	**and**	`&`
slt	`<`	**ult**	`u<`	**sub**	`-`	**urem**	`%u`	**or**	`|`
sle	`<=`	**ule**	`u<=`	**mul**	`*`	**lsh**	`<<`	**xor**	`^`
sgt	`>`	**ugt**	`u>`	**sdiv**	`/`	**ashr**	`u>>`	**neg**	`-`
sge	`>=`	**uge**	`u>=`	**srem**	`%`	**lshr**	`u>>`	**not**	`~`

Table 3.3: Concrete syntax for *rel* when used in comparison predicates and *binop* and *uop* when used in a constant expression.

3.1.3 Type checking

In addition to being syntactically correct, an Alive transformation must be well typed. Like LLVM IR, Alive is strongly typed. Unlike LLVM IR, Alive transformations are parametric over types, meaning they can have multiple valid type assignments. The typing rules for Alive determine whether a particular assignment of concrete types to values (instructions and variables) is valid. If no feasible assignment of types is possible, the transformation is ill typed.

Types

The concrete types in Alive are a subset of the types provided by LLVM IR. These include fixed-width integers (*i. e.*, bit vectors) and floating-point values.[5]

Alive permits integers to have any width. For example, **int** 32 (abbreviated **i32**) is the type of 32-bit integers and **i1** is the type of 1-bit integers (used to represent Boolean values). Each width is considered a different type. Converting an **i1** value to an **i32** requires an explicit conversion.

Alive has five floating-point types, given in Table 3.4. Four are based on IEEE floating-point types: **half**, **float** (single precision), **double**, and **fp128** (quad precision). The last, **x86_fp80**, is an 80-bit floating-point type that represents Intel's extended precision format. It diverges from the IEEE format by including an explicit integer bit, which is normally determined by the class of value: 0 for zero and denormal values, 1 for all other values.

[5]All implementations of Alive also have some support for pointer types. The dialect discussed in this chapter does not exclude pointer types, but also does not include instructions that make use of them beyond **icmp** and **select**.

Type	Width	Exponent	Mantissa
half	16	5	11
float	32	8	24
double	64	11	53
x86_fp80	80	11	64
fp128	128	15	113

Table 3.4: Floating-point types in Alive

Each type has a *width*, corresponding to the number of bits used to represent it. Two distinct types that have the same width are width-equal, *e. g.*, **i16** $=^w$ **half**. Two distinct integer types, or floating-point types, can be ordered by width,[6] *e. g.*, **i16** $<^w$ **i32**. These relations are needed to determine whether certain conversion instructions are correctly typed.

Typing rules

Each value in Alive, including instructions, variables, and constant expressions, can be assigned a type. Predicates are not assigned types, as they always evaluate to true or false, but may include typed subexpressions. An assignment of types to the typed values must conform to the typing rules given in Tables 3.5 to 3.8.

The abstract syntax represents omitted type annotations as $\star$. Each instance of $\star$ may be replaced by any type as necessary. Thus, we can infer **add** $\star\, x, y$: **i32** if x and y are appropriately typed.

Integer literals, the five arithmetic operators in *bop*, unary negation, and the function **abs** may be used for both integer and floating-point types.

Integer literals impose an additional constraint for integer types: the type must be wide enough to represent the literal value. That is, the necessary width for a non-zero integer literal n is at least $\log_2 |n|$. This prevents surprises where Alive attempts to verify a transformation containing a literal constant using a truncated form of that

[6]The floating-point types supported by Alive are naturally ordered by size. This would become less clear if support were added for LLVM's `ppc_fp128` type or for arbitrarily-defined floating-point types, such as a 16-bit representation with 4 exponent bits and 12 mantissa bits.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{VAR} \qquad \frac{\Gamma \vdash x : \tau_1 \quad \tau_1 \overset{conv}{\bowtie} \tau_2}{\Gamma \vdash \mathit{conv}\ \tau_1\ x\ \tau_2 : \tau_2}\ \text{CONV}$$

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash y : \tau \quad \tau \in I}{\Gamma \vdash \mathit{binop}\ \tau\ x, y : \tau}\ \text{BINOP} \qquad \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash y : \tau \quad \tau \in F}{\Gamma \vdash \mathit{fbinop}\ \tau\ x, y : \tau}\ \text{FBINOP}$$

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash y : \tau \quad \tau \in I}{\Gamma \vdash \textbf{icmp}\ \mathit{rel}\ \tau\ x, y : \textbf{int}\ 1}\ \text{ICMP} \qquad \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash y : \tau \quad \tau \in F}{\Gamma \vdash \textbf{fcmp}\ \mathit{frel}\ \tau\ x, y : \textbf{int}\ 1}\ \text{FCMP}$$

$$\frac{\Gamma \vdash x : \textbf{int}\ 1 \quad \Gamma \vdash y : \tau \quad \Gamma \vdash z : \tau}{\Gamma \vdash \textbf{select}\ x, \tau\ y, \tau\ z : \tau}\ \text{SELECT} \qquad \frac{\overline{\Gamma \vdash c_i : \tau_i} \quad \text{SIG}_{\mathit{cfun}}(\tau, \overline{\tau_i})}{\Gamma \vdash \mathit{cfun}(\overline{c_i}) : \tau}\ \text{CFUN}$$

$$\frac{\Gamma \vdash c : \tau \quad \tau \in I \cup F}{\Gamma \vdash \textbf{neg}\ c : \tau}\ \text{NEG} \qquad \frac{\Gamma \vdash c : \tau \quad \tau \in I}{\Gamma \vdash \textbf{not}\ c : \tau}\ \text{NOT}$$

$$\frac{\Gamma \vdash c : \tau \quad \Gamma \vdash d : \tau \quad \tau \in I \cup F}{\Gamma \vdash c\ \mathit{bop}\ d : \tau}\ \text{BOP} \qquad \frac{\Gamma \vdash c : \tau \quad \Gamma \vdash d : \tau \quad \tau \in I}{\Gamma \vdash c\ \mathit{ibop}\ d : \tau}\ \text{IBOP}$$

$$\frac{\tau \in I \cup F}{\Gamma \vdash \mathit{lit} : \tau}\ \text{LIT} \qquad \frac{\tau \in F}{\Gamma \vdash \mathit{flit} : \tau}\ \text{FLIT}$$

$$\frac{\Gamma \vdash c : \tau \quad \Gamma \vdash d : \tau \quad \tau \in I \cup F}{\Gamma \vdash c\ \mathit{rel}\ d\ \text{ok}}\ \text{COMP} \qquad \frac{\overline{\Gamma \vdash c_i : \tau_i} \quad \text{SIG}_{\mathit{pfun}}(\overline{\tau_i})}{\Gamma \vdash \mathit{pfun}(\overline{c_i})\ \text{ok}}\ \text{PFUN}$$

Table 3.5: Typing rules for Alive. The judgement $\Gamma \vdash t : \tau$ indicates that t can be assigned type τ. The judgement $\Gamma \vdash t$ ok indicates that t and its subterms are type-correct.

conv	$\tau_1 \overset{conv}{\bowtie} \tau_2$
bitcast	$\tau_1 =^w \tau_2$
fpext	$\tau_1 \in F \quad \tau_2 \in F \quad \tau_1 <^w \tau_2$
fptosi	$\tau_1 \in F \quad \tau_2 \in I$
fptoui	$\tau_1 \in F \quad \tau_2 \in I$
fptrunc	$\tau_1 \in F \quad \tau_2 \in F \quad \tau_2 <^w \tau_1$
sext	$\tau_1 \in I \quad \tau_2 \in I \quad \tau_1 <^w \tau_2$
sitofp	$\tau_1 \in I \quad \tau_2 \in F$
trunc	$\tau_1 \in I \quad \tau_2 \in I \quad \tau_2 <^w \tau_1$
uitofp	$\tau_1 \in I \quad \tau_2 \in F$
zext	$\tau_1 \in I \quad \tau_2 \in I \quad \tau_1 <^w \tau_2$
ZextOrTrunc	$\tau_1 \in I \quad \tau_2 \in I$

Table 3.6: Relations for conversion instructions

cfun	$\text{SIG}_{cfun}(\tau, \overline{\tau_i})$
abs	$\tau = \tau_1 \quad \tau \in I \cup F$
computeKnownOneBits	$\tau = \tau_i \quad \tau \in I$
computeKnownZeroBits	$\tau = \tau_i \quad \tau \in I$
ComputeNumSignBits	$\tau \in I \quad \tau_1 \in I$
countLeadingZeroes	$\tau \in I \quad \tau_1 \in I$
countTrailingZeroes	$\tau \in I \quad \tau_1 \in I$
fpext	$\tau \in F \quad \tau_1 \in F \quad \tau_1 <^w \tau$
fpMantissaWidth	$\tau \in I \quad \tau_1 \in F$
fptosi	$\tau \in I \quad \tau_1 \in F$
fptoui	$\tau \in I \quad \tau_1 \in F$
fptrunc	$\tau \in F \quad \tau_1 \in F \quad \tau <^w \tau_1$
log2	$\tau \in I \quad \tau_1 \in I$
max	$\tau = \tau_1 = \tau_2 \quad \tau \in I$
min	$\tau = \tau_1 = \tau_2 \quad \tau \in I$
sext	$\tau \in I \quad \tau_1 \in I \quad \tau_1 <^w \tau$
sitofp	$\tau \in F \quad \tau_1 \in I$
trunc	$\tau \in I \quad \tau_1 \in I \quad \tau <^w \tau_1$
uitofp	$\tau \in F \quad \tau_1 \in I$
umax	$\tau = \tau_1 = \tau_2 \quad \tau \in I$
umin	$\tau = \tau_1 = \tau_2 \quad \tau \in I$
width	$\tau \in I \quad \tau_1 \in I$
zext	$\tau \in I \quad \tau_1 \in I \quad \tau_1 <^w \tau$

Table 3.7: Constant function signatures

pfun	$\text{SIG}_{pfun}(\overline{\tau_i})$
CannotBeNegativeZero	$\tau_1 \in F$
fpIdentical	$\tau_1 = \tau_2 \quad \tau_1 \in F$
fpInteger	$\tau_1 \in F$
isPowerOf2	$\tau_1 \in I$
isPowerOf2OrZero	$\tau_1 \in I$
isShiftedMask	$\tau_1 \in I$
isSignBit	$\tau_1 \in I$
MaskedValueIsZero	$\tau_1 = \tau_2 \quad \tau_1 \in I$
WillNotOverflowSignedAdd	$\tau_1 = \tau_2 \quad \tau_1 \in I$
WillNotOverflowSignedSub	$\tau_1 = \tau_2 \quad \tau_1 \in I$
WillNotOverflowSignedMul	$\tau_1 = \tau_2 \quad \tau_1 \in I$
WillNotOverflowUnsignedAdd	$\tau_1 = \tau_2 \quad \tau_1 \in I$
WillNotOverflowUnsignedSub	$\tau_1 = \tau_2 \quad \tau_1 \in I$
WillNotOverflowUnsignedMul	$\tau_1 = \tau_2 \quad \tau_1 \in I$
WillNotOverflowUnsignedShl	$\tau_1 = \tau_2 \quad \tau_1 \in I$

Table 3.8: Predicate function signatures. Functions not listed here have unconstrained signatures.

constant.[7] It is unusual for Alive transformations to use literals other than 1, 0, and −1, so this restriction rarely arises.

Type ambiguity

Certain expressions, such a literal constants and the function **width**, do not have a uniquely-determined type even when all variables have been assigned types. Instead, the context of the expression determines its type. However, comparison predicates and some constant and predicate functions do not fully constrain their arguments. Thus, in the expression `C + trunc(zext(C))`, the type of `zext(C)` is only constrained to be wider than the type of `C`.

An expression has an *ambiguous type* if it has more than one type consistent with any type assignment for the values in the source. If a transformation contains an ambiguously typed expression, then its implementation would have to arbitrarily choose a type for that expression when testing the precondition or creating the target. To avoid this situation, Alive forbids most ambiguous types.

Ambiguously typed comparison predicates are permitted, in order to allow predicates of the form `width(%x) u> 1`. If a comparison predicate has an ambiguous type, Alive fixes it to a *default type*, usually **i64**. This reflects the implementation of LLVM, where the equivalent of width returns an unsigned 64-bit integer.

If an ambiguously typed constant expression is used in multiple places, at least one of which determines its type, it can be named using a symbolic constant to force its uses to have the same type, as in Figure 3.4.

3.1.4 DAG Representation

The abstract and concrete syntaxes for Alive describe the source and target patterns as a list of statements. The rules for scoping ensure that instruction results are not referenced prior to their definition, thus preventing circular dependencies. However, this

[7] Note that 8 and −8 both require at least four bits, but in the **i4** case will have the same representation. Some versions of Alive require an additional bit for positive literals, so that for example $8 \neq -8$ holds for all feasible types, but this proved surprising in other ways, such as 1 requiring two bits.

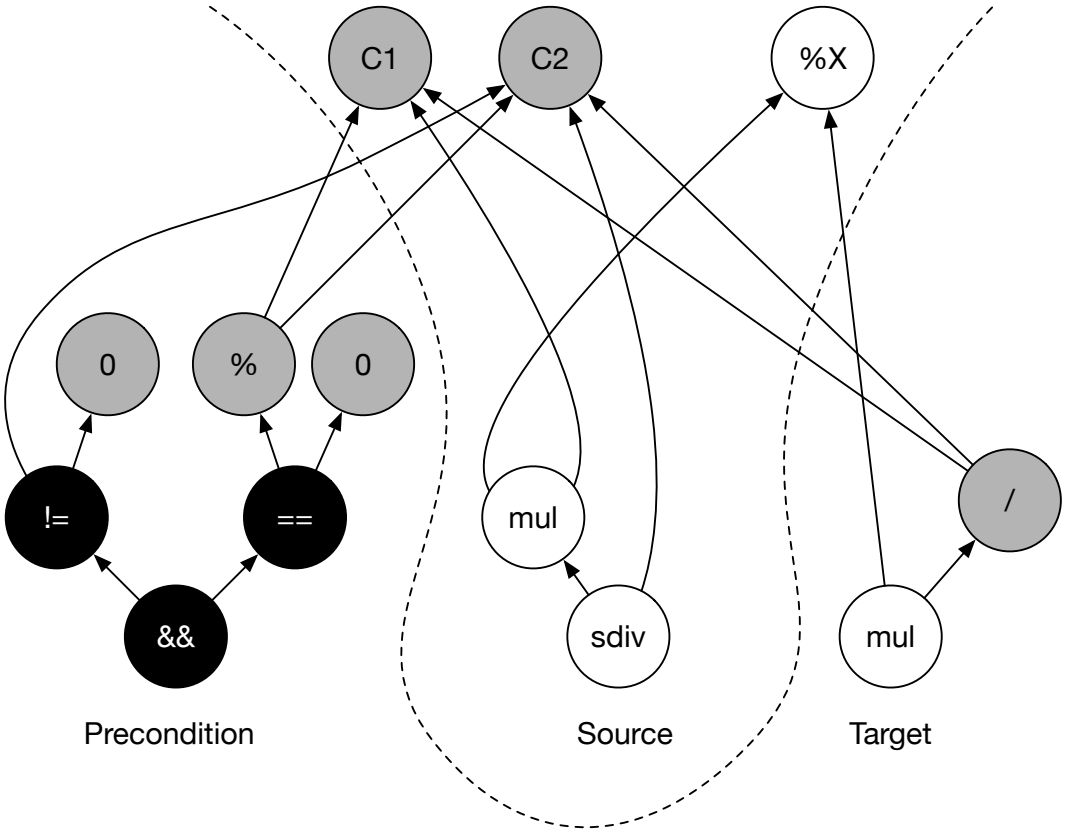


Figure 3.5: The optimization from Figure 1.2 represented as a DAG. Instructions and input variables are in white, symbolic constants and constant expressions are in gray, and predicates are in black. The dashed lines indicate the source sub-DAG.

presentation can be misleading: the source pattern presents a single, ordered block of instructions, but instructions matched by the source pattern may occur in different basic blocks, non-matched instructions may be interleaved with the matched instructions, and instructions that do not refer to each other may occur in any order.

The source and target describe a code fragment in SSA form, so it is also possible to represent them as a directed, acyclic graph (DAG). The source, target, and precondition all refer to certain nodes in common, such as the free variables, so the transformation as a whole can be represented as a DAG with three root nodes, representing to the roots of the source and target and the precondition. Figure 3.5 shows the DAG representation of the optimization from Figure 1.2.

The nodes of the DAG representation correspond to instructions, variables, constant expressions, and predicates. The arcs represent data dependency: an arc from node x

to node y indicates that the computation of x depends on the result of y. The out-going arcs from a node are ordered, corresponding to the order of parameters for an instruction, function, or operator. Leaf nodes, with no out-going arcs, represent the free variables and symbolic constants. Root nodes, with no in-coming arcs, represent the roots of the source and target and the precondition.[8]

Nodes can be categorized according to the sub-languages of Alive, indicated by shading in Figure 3.5. Arcs within a sub-language are common, as are arcs from instructions and predicates to constant expressions. Arcs from predicates and constant expressions indicate an indirect dependence: because predicates and constant expressions are evaluated at compile time and instructions are not evaluated until run time, constant expressions and predicates cannot depend on the result of an instruction or input variable. Instead, these arcs indicate that the predicate or constant expression depends on a static analysis of the instruction or input variable.

The DAG for a transformation can be divided into three sub-DAGs, representing the source, target, and precondition. The primary sub-DAG is the source DAG, which includes all nodes that the source root depends on, transitively. The source sub-DAG represents the fragment of a program that the optimization will transform. It may contain only instructions and free variables; the scoping rules guarantee that it contains *all* free variables. Similarly, the sub-DAGs for the target and precondition are the transitive dependencies of the target and precondition roots. Both may contain constant expressions. Only the precondition sub-DAG may include predicates.

The target sub-DAG may include instructions not present in the source sub-DAG: these are new instructions that will be created after the precondition is evaluated. The precondition sub-DAG will only refer to instructions in the source sub-DAG. Named constant expressions may be shared between the precondition and target sub-DAGs.

Two transformations may present the same instructions in a different order while remaining consistent with the scoping rules. These will have the same DAG representation, with the orders of instructions corresponding to different linearizations of the

[8]If arcs were drawn in the reverse direction, so that data flows in the direction of the arrow, these roots would be leaf nodes and the free variables would be root nodes.

DAG. When using the DAG representation, the order in which statements are presented can be ignored, making it simpler to reason about Alive transformations.

3.2 Verification

An important design goal for Alive is automated verification of transformations. That is, it should be possible to determine the correctness of an Alive-specified transformation without requiring user input beyond the specification itself. A correct transformation will have two properties. The first is that applying the transformation must not change the behavior of a program. Because the semantics of LLVM has some nondeterministic aspects, it is possible for a single program to have multiple possible behaviors. In those cases, transforming a program must not introduce new possible behaviors, but may eliminate some behaviors (*i. e.*, the target must refine the source).

The second, less obvious property is that applying the transformation itself must be a well-defined process. The precondition and target of a transformation may include constant expressions. These will be evaluated by the compiler when it tests the precondition and creates the target instructions. The constant language for Alive includes four division-like operators, which are undefined when their divisors are zero. If the compiler divides by zero while attempting to apply a transformation, it may crash.

Determining whether a transformation is correct requires checking whether the target refines the source for all possible valuations of its free variables. This can be accomplished by expressing the correctness conditions as queries to a solver for satisfiability modulo theories (SMT), but Alive transformations may have multiple type assignments while terms in an SMT query must have fixed types. These correctness conditions must also model LLVM's three forms of undefined behavior and its imprecise dataflow analyses.

3.2.1 Structure of Alive encodings

Using an SMT solver to check the correctness of an Alive transformation requires encoding its correctness conditions as propositions in a logic involving one or more theories,

such as the SMT theories of bit vectors, floating-point arithmetic, and arrays. These propositions are not type parametric, so each feasible type assignment must be encoded and checked for correctness separately.[9]

The correctness conditions are quantified logical propositions generated from the encodings of the source, target, and precondition of a transformation, which are encoded by recursively encoding individual instructions, variables, constant expressions, and predicates. Quantification applies to the correctness conditions as a whole, so the encodings for individual parts of a transformation will include free variables.

Alive expressions are encoded as SMT expressions and propositions describing their compile-time behavior (σ) and run-time behavior (δ, ρ, ι). Subscripts are used to indicate the expression being encoded. By convention, the roots of the source, target, and precondition are indicated with $-_s$, $-_t$, and $-_p$.

The value computed by an instruction, variable, or constant x is written ι_x. This will be an arithmetic expression of an SMT sort corresponding to the type of x. For a predicate p, ι_p is a logical formula indicating whether p holds. By convention, ι_p is written ϕ when p is the precondition.

3.2.2 Encoding compile-time behavior

The target and precondition of an Alive transformation may include constant expressions, which represent computation performed by the compiler, rather than the program being transformed. Constant expressions may include four distinct division-like operators that are not defined when used with integer types and their divisors are zero, and their corresponding implementations in C++ are similarly undefined. To keep the behavior of the compiler defined, the precondition must prevent any evaluation of undefined constant expressions. To distinguish undefined behavior in the compiler from undefined behavior in the program being transformed, the former is called *unsafe* behavior.

[9] Alive transformations appear to have an inductive property where, for some type assignment, assignments using concrete types of the same size or larger will be all correct (or all incorrect). Unfortunately, the specific details of this property are unknown.

The safety condition (σ) for an Alive expression indicates that the expression does not have unsafe behavior or depend on a value that has unsafe behavior. Expressions that do not include a constant expression or depend on one, such as the source, will have trivial safety conditions (*i. e.*, $\sigma_s \equiv \top$, where $\top$ is always satisfied).

3.2.3 Encoding run-time behavior

The source and target of a transformation describe a fragments of LLVM IR code. In order to check correctness, it is necessary to model their run-time behavior. These behaviors include the three forms of undefined behavior defined by the LLVM semantics.

Certain instructions in LLVM are not defined for all inputs. For example, **udiv** $\star\ x, y$ is undefined when $y = 0$. To model this, each expression in Alive has a *well-defined* condition (δ). An instruction will have undefined behavior if and only if δ is unsatisfied. The well-defined condition is trivial for non-instruction expressions. If an instruction has undefined behavior, then all subsequent instructions are undefined. Alive instructions are not totally ordered, but an instruction that refers to another instruction is guaranteed to occur after it. Thus, a particular instruction is well-defined only if its arguments are well-defined.

Certain instructions may return a special poison value for some inputs. For example, **add nsw** $\star\ x, y$ returns poison if $x + y$ causes a signed overflow. To model this, each expression in Alive has a *poison-free* condition (ρ). An instruction returns poison if and only if ρ is unsatisfied. The poison-free condition is trivial for non-instruction expressions. Instructions usually return poison if any of their arguments are poison.

The well-defined and poison-free conditions for an expression must be satisfied for its value to be meaningful. If δ_x is not satisfied for some input, then x may have any behavior, including returning a poison value or returning any possible value in its type. Because any undefined behavior in the source or target will cause the source or target as a whole to be undefined, it is sufficient to note when undefined behavior has occurred. Returning a poison value is similarly non-deterministic, as x is free to return any possible value if ρ_x is not satisfied. This must be modeled, as an instruction returning a poison value may violate the well-defined condition for a subsequent instruction.

Consider this program fragment:

```
%a = and %x, 1
%b = udiv %y, %a
```

Ignoring poison, `%b` is always well-defined (assuming all prior instructions are also well-defined): the **and** with 1 guarantees that `%a` is non-zero. However, if `%x` is poison, then `%a` will return poison, and one of its possible behaviors is returning 0. Thus, the well-defined conditions for `%b` require `%x` to be poison-free.

3.2.4 Encoding `undef` values

A computation that is well-defined and poison-free may still be non-deterministic in LLVM. The `undef` value and instructions that return `undef` have a set of behaviors that include returning all normal values of a type, but not returning poison or having true undefined behavior. An instruction with an `undef` input may non-deterministically choose which value to use, creating a new set of behaviors.

Note that it is not sufficient to indicate whether a value is `undef` with a flag. It is possible that an instruction will return a value chosen non-deterministically from a subset of possible values. For example, the possible return values of `mul undef, 2` include only even numbers.

If an instruction has multiple arguments with non-deterministic values, it may select each argument independently—even if those arguments refer to the same instruction. If x has two behaviors, then $x \oplus x$ may have up to four.

Rather than represent ι_x as a set, it is encoded as an arithmetic expression that may contain additional free variables. In particular, `undef` will be encoded as a fresh variable u that has not been used for any other value in the transformation. In order to ensure independence, a value which is referenced multiple times must use different variables each time. For example, when encoding these instructions

```
%a = mul undef, 2
%b = xor %a, %a
```

the proper encoding will have $\iota_b = 2u_1 \oplus 2u_2$ (where $\oplus$ is exclusive or). Note that the two references to ι_a are replaced by expressions using different fresh variables to represent the `undef`.

The correctness conditions require that the set of possible behaviors exhibited by the target be a subset of the possible behaviors of the target. That is, for every choice that may be made for the target, there must be a choice for the source that makes them equal. This implies that these variables will be quantified separately, based on whether they occur in the source or the target. To allow for this, the encoding of an Alive expression x also includes a set $\mathcal{Q}_x$ of free variables used for encoding `undef` and `undef`-like behavior.

3.2.5 Encoding "undefined results"

The LLVM Language Reference describes several circumstances where an instruction has "an undefined result", a phrase that is not explicitly defined but is used in contrast to "undefined behavior" in some cases. This could be interpreted as returning an `undef`-like value or as returning poison. The former seems likely to be the original intent, but the latter leads to simpler correctness conditions.

Let ψ be the condition for an instruction to have a defined result. Under an `undef`-like interpretation, the instruction will return its usual result when ψ is satisfied, and a fresh variable $u \in \mathcal{Q}$ otherwise. Under the poison-like interpretation, ψ is part of the poison-free condition.

For example [83], the LLVM's left shift instruction **shl** has an undefined result if the shift amount exceeds the width of the result. Under the `undef`-like interpretation, this would be encoded as

$$\iota = \begin{cases} \iota_x << \iota_y & \text{if } \iota_y <_u w \\ u & \text{if } \iota_y \geq_u w \end{cases} \tag{3.2}$$

$$\rho \equiv \rho_x \wedge \rho_y \tag{3.3}$$

$$\delta \equiv \delta_x \wedge \delta_y \tag{3.4}$$

$$\mathcal{Q} = \mathcal{Q}_x \wedge \mathcal{Q}_y \wedge \{u\}, \tag{3.5}$$

```
  %Op0 = shl 1, %Y
  %r   = mul %Op0, %Op1
=>
  %r   = shl %Op1, %Y
```

(a)

```
Pre: C == 0.0
  %y = fsub nnan ninf C, %x
  %z = fadd %y, %x
=>
  %z = 0
```

(b)

Figure 3.6: Two transformations that are correct under the poison-like encoding, but incorrect under the `undef`-like encoding

where x and y are the arguments, w is the bit width, and u is a fresh variable. Under the poison-like interpretation, it would be encoded as

$$\iota = \iota_x << \iota_y \tag{3.6}$$

$$\rho \equiv \rho_x \wedge \rho_y \wedge \iota_y <_u w \tag{3.7}$$

$$\delta \equiv \delta_x \wedge \delta_y \tag{3.8}$$

$$\mathcal{Q} = \mathcal{Q}_x \wedge \mathcal{Q}_y. \tag{3.9}$$

While a strong argument can be made that the `undef`-like encoding is the intended meaning of the language reference, some parts of LLVM implicitly assume poison-like behavior. Both the transformations in Figure 3.6 are based on code in InstCombine, and both are correct under the poison encoding but not under the undef encoding. In Figure 3.6(a), the if the shift amount `%Y` is too large and `%Op1` is zero, the source and target are both poison under the poison encoding, but the source is zero and the target is `undef` under the `undef`-like encoding.[10] In Figure 3.6(b), if `%x` is NaN, the source will be poison under the poison-like encoding, but NaN under the `undef`-like encoding. A poison value can be refined to 0, but a NaN cannot. Under the `undef`-like encoding, the **nnan** attribute on `%y` is not enough to prevent `%z` from being NaN.[11]

Alive-NJ is designed to support multiple encodings, which users may choose from when verifying a transformation. The `undef`-like encoding is used by default.

[10]The `undef`-like encoding could be fixed here by adding an exception that zero shifted by any number of places always yields zero.

[11]This transformation would also be correct if the **nnan** attribute were interpreted as making a guarantee about `%x`, not just about `%y`. That is, if any use of `%x` has a **nnan** attribute, then `%x` can be considered non-NaN everywhere it is referenced. This sort of non-local encoding is possible, but it is not clear whether it has any advantage over the poison encoding.

3.2.6 Encoding data-flow analyses

Several Alive predicates, such as **WillNotOverflowSignedAdd**, and constant functions, such as **ComputeNumSignBits**, rely on static analyses performed by LLVM. Several of these analyses are imprecise, meaning that they over- or under-approximate the correct result. For example, a *must-analysis* is satisfied when the compiler can prove that a certain condition holds. A negated must-analysis is true when the analysis fails, not necessarily when the condition is false.

An imprecise analysis is encoded in two parts: a variable representing the analysis result, and an *auxiliary condition* (χ) that relates the analysis result to the analyzed values.

For example, consider a transformation with the precondition `!isPowerOf2(%a)`. A naïve encoding of this precondition would be

$$\phi \equiv \neg[a \,\&\, (a-1) = 0 \wedge a \neq 0], \tag{3.10}$$

writing & for bit-wise and. This assumes that the transformation will not apply if it is possible for `%a` to be a power of two. However, the precondition actually requires that the compiler fail to prove `%a` to be a power of two, which does not constrain `%a` in any way. Under the correct encoding,

$$\phi \equiv \neg b \tag{3.11}$$

$$\chi_p \equiv b \implies a \,\&\, (a-1) = 0 \wedge a \neq 0, \tag{3.12}$$

the precondition constrains b to be false, in which case the auxiliary condition is consistent with any value of a. (Both encodings have been simplified by ignoring poison. See Section 3.3.8 for the full encoding.)

Imprecise constant functions are encoded similarly. A fresh variable represents the function result, and the auxiliary condition defines which values the result may have (see Section 3.3.7).

Imprecise analyses should give the same result when a predicate or function is repeated. That is, the precondition `isPowerOf2(%a) && !isPowerOf2(%a)` should never

be satisfiable, as it would require the analysis to fail and succeed for the same instruction. This can be approximated by remembering the variables used to encode each imprecise function for a particular argument list, and re-using that variable if the combination occurs again.

3.2.7 Correctness conditions

In order to be correct, an we must show that the transformation is safe to execute and that it does not introduce new behaviors. These correctness conditions can be defined using the encodings of the source, target, and precondition.

The encodings have six parts. The value (ι) gives the result of evaluating the source, target, or precondition (written ϕ). The safety condition (σ) indicates when the compile-time computation in the target or precondition can be performed safely. The well-defined (δ) and poison-free (ρ) conditions limit the possible behaviors of the source and target. The set $\mathcal{Q}$ gives the variables used to encode non-deterministic results from `undef` and `undef`-like behavior. Finally, an auxiliary condition χ limits the range of variables used to encode imprecise analyses.

Let $\mathcal{I}$ be the set of variables representing input variables and $\mathcal{C}$ be the set of variables representing symbolic constants. Let $\mathcal{P}$ be the variables used to represent imprecise analyses. These, along with $\mathcal{Q}_s$, $\mathcal{Q}_t$, and $\mathcal{Q}_p$, contain all the free variables used in the encodings.

Of these sets, only $\mathcal{Q}_s$ requires special handling. The remaining sets can be treated collectively:

$$\mathcal{V} = \mathcal{I} \cup \mathcal{C} \cup \mathcal{P} \cup \mathcal{Q}_t \cup \mathcal{Q}_p. \tag{3.13}$$

The correctness conditions are universally quantified over the variables in $\mathcal{V}$, but existentially quantified over the variables in $\mathcal{Q}_s$. If $\mathcal{Q}_s$ is non-empty (*i. e.*, the source is potentially non-deterministic), it is sufficient to show that every behavior of the target is a refinement of some behavior in the source. If $\mathcal{Q}_s$ were universally quantified, it would require every behavior of the target to refine *every* behavior of the source, which would be impossible if the source had more than one well-defined, poison-free behavior.

The correctness conditions can be expressed as a single constraint, but we present them as five constraints for ease of understanding. Testing these constraints individually allows for better error messages.

1. The precondition must always be safe to evaluate.

$$\forall \mathcal{V} : \chi_p \implies \sigma_p \tag{3.14}$$

2. When the precondition holds, the target must be safe to create.

$$\forall \mathcal{V} : \chi_p \wedge \chi_t \wedge \phi \implies \sigma_t \tag{3.15}$$

3. When the precondition holds and the source is well-defined, the target must be well-defined.

$$\forall_{\mathcal{V}} \exists_{\mathcal{Q}_s} : \chi_p \wedge \chi_t \wedge \phi \wedge \delta_s \implies \delta_t \tag{3.16}$$

4. When the precondition holds and the source is well-defined and poison-free, the target must be poison-free.

$$\forall_{\mathcal{V}} \exists_{\mathcal{Q}_s} : \chi_p \wedge \chi_t \wedge \phi \wedge \delta_s \wedge \rho_s \implies \rho_t \tag{3.17}$$

5. When the precondition holds and the source is well-defined and poison-free, the source and target must compute the same value

$$\forall_{\mathcal{V}} \exists_{\mathcal{Q}_s} : \chi_p \wedge \chi_t \wedge \phi \wedge \delta_s \wedge \rho_s \implies \iota_s = \iota_t \tag{3.18}$$

Justifying separate existential quantifiers

One might expect the correctness condition to be given as a single proposition, such as

$$\forall_{\mathcal{V}} \exists_{\mathcal{Q}_s} : \chi_p \implies \sigma_p \wedge [\chi_t \wedge \phi \implies \sigma_t \wedge (\delta_s \implies \delta_t) \wedge (\delta_s \wedge \rho_s \implies \rho_t \wedge \iota_s = \iota_t)]. \tag{3.19}$$

This can be rewritten as a conjunction of the five correctness conditions given above, but it may not be clear whether the reverse is true because of the existential quantifiers. In general, $(\exists_x : P) \wedge (\exists_x : Q)$ does not imply $\exists_x : P \wedge Q$. This suggests the possibility of a transformation that only satisfies the correctness conditions by choosing different valuations for $\mathcal{Q}_s$ in the different conditions.

However, the structure of the Alive encoding prevents this. First, the variables in $\mathcal{Q}_s$ only occur in δ_s, ρ_s, and ι_s. Beacause of the way `undef`-like values are encoded, any reference to source values in the target or precondition will use fresh variables collected in $\mathcal{Q}_t$ or $\mathcal{Q}_p$, respectively. Next, because δ_s and ρ_s are logical formulae, the distinction between two valuations of $\mathcal{Q}_s$ which both satisfy or do not satisfy them is irrelevant. If, for a particular valuation of $\mathcal{V}$, there are multiple valuations of $\mathcal{Q}_s$ such that $\neg\delta_s$, it does not matter which one is chosen because any would be sufficient to satisfy conditions 3–5. In contrast, the valuation chosen for $\mathcal{Q}_s$ will affect the value of ι_s, but ι_s occurs only under one existential quantifier, so there is no possibility of choosing inconsistently.

Therefore, it is not possible for the encoding of an Alive transformation to satisfy conditions 1–5 without also satisfying Equation (3.19).

Why $\mathcal{Q}_p$ is universally quantified

The need for $\mathcal{Q}_p$ might be surprising, as `undef` does not generally occur in the precondition. However, some instructions may have `undef`-like behavior, and their results can be analyzed by some predicates. Because each direct or indirect reference to an `undef`-like value receives a fresh variable, the variables used to encode the precondition will be distinct from those in $\mathcal{Q}_s$ or $\mathcal{Q}_t$.

It might seem reasonable for $\mathcal{Q}_p$ to be existentially quantified: after all, any analysis in the precondition that relies on `undef` should be able to choose a convenient value. But existential quantification actually allows the correctness check to choose the value, not the (simulated) analysis. In fact, we need to show that the optimization is correct no matter what value the analysis chooses for the `undef`.

Consider this (contrived) transformation:

```
Pre: isPowerOf2(%a)
  %a = shl i4 1, %x
  %b = select 1, %x, %a
  %c = icmp ult %b, 4
=>
  %c = true
```

Does the precondition permit `%x` to be 4? Under the `undef`-like interpretation, the left shift would overflow and `%a` would be `undef`, so the analysis behind `isPowerOf2` would be free to pick a power of two and say the condition is satisfied. (In fact, the analysis as implemented always returns true given a left-shift of one.)

The value check for this optimization simplifies to:

$$\forall_{x,u} : \begin{cases} pow2(1 << x) \implies 1 = 1 & \text{if } x <_u 4 \\ pow2(u) \implies 0 = 1 & \text{if } x \geq_u 4 \end{cases} \tag{3.20}$$

It is easy to find a counter-example ($x = 4 \ \wedge \ u = 1$), so we can conclude this optimization is incorrect.

In contrast, if $\mathcal{Q}_p$ were existentially quantified, the case where $x \geq_u 4$ would become $\exists_u : pow2(u) \implies 0 = 1$, which is true.

Why χ_p occurs in the precondition safety check

As χ_p is used to constrain run-time inputs under the assumption that an analysis has succeeded, it may be surprising that it is needed to check the safety of the precondition, a property that relies only on compile-time data. However, the safety condition may also include variables from $\mathcal{P}$, and χ_p may limit their range.

Consider the expression `C1 % (1 << ComputeNumSignBits(%a))`, which might occur in a precondition. The safety condition for this expression is $1 << b \neq 0$. If b is unconstrained, then this condition can be violated by choosing b larger than the bit width of `C1`, as shifts in SMT return zero when the shift amount exceeds the bit width. However, the auxiliary condition for this expression is $b \leq_u \mathsf{signBits}(\iota_a)$, where signBits will never return a value larger than the bit width of `%a`. Therefore, this expression is only unsafe if `%a` has a larger width than `C1`.

The auxiliary conditions for the target (χ_t) occur in the target safety check for the same reason.

Expressing the conditions as SMT queries

The correctness conditions were given above as statements that must be proven true in order to show that an optimization is correct. However, SMT solvers operate by

finding variable assignments that satisfy a formula. In order to use an solver to prove the correctness conditions, we negate them and ask for counter-examples.

Thus, the actual queries given to the solver are

1. $\exists_{\mathcal{V}} : \chi_p \wedge \neg\sigma_p$
2. $\exists_{\mathcal{V}} : \chi_p \wedge \chi_t \wedge \phi \wedge \neg\sigma_t$
3. $\exists_{\mathcal{V}}\forall_{\mathcal{Q}_s} : \chi_p \wedge \chi_t \wedge \phi \wedge \delta_s \wedge \neg\delta_t$
4. $\exists_{\mathcal{V}}\forall_{\mathcal{Q}_s} : \chi_p \wedge \chi_t \wedge \phi \wedge \delta_s \wedge \rho_s \wedge \neg\rho_t$
5. $\exists_{\mathcal{V}}\forall_{\mathcal{Q}_s} : \chi_p \wedge \chi_t \wedge \phi \wedge \delta_s \wedge \rho_s \wedge \iota_s \neq \iota_t$

3.3 Encoding Expressions for Verification

To check the correctness of a transformation, its source, target and precondition must be encoded into logical formulae and arithmetic expressions. The encoding process is recursive: the encoding of an instruction or constant expression will include the encodings of its arguments. In terms of the abstract syntax, each production of *val*, *inst*, and *pre* is encoded, with bound variables using the encoding of the expression they are assigned. In terms of the DAG representation, each node is encoded.

Encoding is performed with respect to a specific type assignment. The encoding of an expression contains three logical propositions indicating whether the expression is safe (σ), well-defined (δ), and poison-free (ρ). Additional requirements are given by the auxiliary conditions (χ). The value computed by an instruction, variable, or constant (ι) will be an SMT expression of a sort corresponding to the Alive expression. The value computed by the precondition (ϕ) will be another logical proposition. Finally, the set $\mathcal{Q}$ contains any variables used to represent non-deterministic behavior from `undef` or `undef`-like operations.

3.3.1 Encoding types

Alive expressions that return a value have a corresponding type. These will be encoded as expressions in SMT bit vector and floating-point arithmetic of a corresponding SMT

sort. For Alive integers, the corresponding sort is a bit vector of the same width. For the floating-point types, the corresponding sort is a floating-point type with the same exponent and mantissa size (see Table 3.4).[12]

While **x86_fp80** is an 80-bit floating point type, its exponent and mantissa correspond to a 79-bit floating-point sort in SMT. This is because it diverges from the IEEE format by including an explicit integer bit, which is 0 for zero and denormal values, and 1 for all other values. Negating the integer bit results in pseudo-values and "unnormals". These non-canonical values are not produced by processors since the Intel 80386, and have no processor-independent interpretation in the LLVM language standard. For this reason, Alive assumes that the integer bit is always set correctly.

SMT floating-point types include a single value for NaN, while the IEEE format used by LLVM and Alive contains many distinct NaNs. This has two consequences. First, the correctness conditions may allow a transformation to change one NaN result into another. The IEEE standard and LLVM semantics do not mandate that any particular NaN be produced by an arithmetic operation, except that an operation with receives one or more NaN arguments should use one of those as its result, so it is not clear whether such a transformation should be considered incorrect. Second, the **bitcast** instruction cannot correctly model conversions to and from floating-point types, as they will use a canonical NaN value instead of preserving the exact bit pattern. In particular, converting an integer to a floating-point value and back is not an identity function, as required by the LLVM language reference.

3.3.2 Encoding variables

Alive has two kinds of variables: input variables (*e. g.*, `%x`), which represent arbitrary values, and symbolic constants (*e. g.*, `C1`), which represent in-line constants that will be known at compile time. Their values, ι, are encoded as fresh SMT variables of the appropriate sort.

[12]While Alive assumes that floating-point types are formatted according to IEEE-754, a careful reading of the LLVM Language Reference shows that only the widths of the floating-point types are guaranteed.

Instruction	δ
udiv int $w\,x, y$	$\delta_x \wedge \delta_y \wedge \rho_y \wedge \iota_y \neq 0$
urem int $w\,x, y$	$\delta_x \wedge \delta_y \wedge \rho_y \wedge \iota_y \neq 0$
sdiv int $w\,x, y$	$\delta_x \wedge \delta_y \wedge \rho_y \wedge \iota_y \neq 0 \wedge \left(\iota_y \neq -1 \vee \rho_x \wedge \iota_x \neq -2^{w-1}\right)$
srem int $w\,x, y$	$\delta_x \wedge \delta_y \wedge \rho_y \wedge \iota_y \neq 0 \wedge \left(\iota_y \neq -1 \vee \rho_x \wedge \iota_x \neq -2^{w-1}\right)$

Table 3.9: Well-defined conditions for selected instructions. The minimum signed w-bit integer is -2^{w-1}.

Instruction	Defined-result condition
shl int $w\,x, y$	$\iota_y <_u w$
ashr int $w\,x, y$	$\iota_y <_u w$
lshr int $w\,x, y$	$\iota_y <_u w$
fptrunc $\tau_1\,x$ **to** τ_2	$\mathsf{fmin}(\tau_2) \leq \iota_x \leq \mathsf{fmin}(\tau_2) \vee \mathsf{isInfinite}(\iota_x) \vee \mathsf{isNaN}(\iota_x)$
fptosi $\tau_1\,x$ **to int** w	$-2^{w-1} - 1 < \iota_x < 2^{w-1}$
fptoui $\tau_1\,x$ **to int** w	$0 \leq \iota_x < 2^w$
sitofp $\tau_1\,x$ **to** τ_2	$\mathsf{fmin}(\tau_2) \leq \iota_x \leq \mathsf{fmax}(\tau_2)$
uitofp $\tau_1\,x$ **to** τ_2	$\iota_x \leq_u \mathsf{fmax}(\tau_2)$

Table 3.10: Defined result conditions, where $\iota_x \in \tau$ indicates that ι_x can be exactly represented in τ and $\mathsf{fmin}(\tau)$ and $\mathsf{fmax}(\tau)$ are the minimum and maximum finite floating-point values in τ. The bounds for **fptosi** and **fptoui** reflect that the argument will be rounded towards zero. Range checks that are statically known to succeed (*e. g.*, because the upper bound exceeds the maximum representable value) may be omitted.

An input variable may contain poison. To represent this, its poison-free condition, ρ, is encoded as a fresh SMT Boolean variable. A symbolic constant is never poison, so its poison-free condition is trivial.

3.3.3 Encoding arithmetic and conversion instructions

With a few exceptions, Alive's arithmetic and conversion instructions correspond directly to analogous operations in SMT bitvector or floating-point logic. The four integer division operations have special well-defined conditions, given in Table 3.9. The shift operations and floating-point conversions have special defined-result conditions (see Section 3.2.5), which are given in Table 3.10. Several arithmetic instructions may be modified with attributes, which are discussed in the next section.

The **frem** instruction computes the remainder after dividing two floating-point values. In LLVM, the result of **frem** will have the same sign as the second argument, which does not correspond to the floating-point remainder function used by SMT and

$$\mathsf{fpMod}(x,y) = \begin{cases} z & \text{if } \mathsf{isPos}(x) \wedge \mathsf{isPos}(z) \\ |y| + z & \text{if } \mathsf{isPos}(x) \wedge \mathsf{isNeg}(z) \\ -z & \text{if } \mathsf{isNeg}(x) \wedge \mathsf{isPos}(z) \\ -(|y| + z) & \text{if } \mathsf{isNeg}(x) \wedge \mathsf{isNeg}(z) \end{cases}$$
$$z = \mathsf{fpRem}(|x|, |y|)$$

Figure 3.7: Computing fpMod, which has the same sign as its second argument, using the IEEE fpRem function provided by SMT

IEEE floating-point. Figure 3.7 shows how to compute an **frem**-like function using the SMT-provided fpRem function and some sign manipulation.

The **fpext** and **fptrunc** instructions are both encoded using SMT's overloaded conversion to floating-point function. The **fptrunc** instruction has undefined results for finite arguments outside the finite range of the target type. Arguments inside the range that cannot be represented exactly are rounded.

The **fptosi** and **fptoui** instructions round towards zero (*i. e.*, truncate), but have undefined results when the result is too large to be represented in the target type. In Table 3.10, converting any negative value to an unsigned integer has undefined results. The lower bound could plausibly be weakened to $-1 < \iota_x$. If the argument is NaN, this encoding has an undefined result. It could be argued that this would cause a floating-point exception, and therefore should be encoded as undefined behavior.

The **bitcast** instruction converts a value to another type without changing any of the underlying bits. In SMT, floating-point types are not necessarily represented as bit vectors containing IEEE-formatted values, so the conversion must be implemented using SMT functions. Because SMT floating-point types include only a single NaN value, conversion from integers will lose information and conversion to integers is arbitrary or non-deterministic, depending on the method used. For this reason, the claims Alive can make about transformations involving **bitcast** are limited.

Assumptions about floating-point arithmetic

The LLVM Language Reference [2] refers to the IEEE-754 standard for floating-point arithmetic [3] in several places, but never explicitly requires that its floating-point

instructions conform to that standard. Without a formal specification for the meaning of floating-point operations, no formal verification is possible. However, transformations in LLVM tend to assume IEEE semantics, which is understandable given the increasing rarity of hardware implementing non-IEEE floating-point. Thus, Alive limits itself to verifying transformations under the assumption of IEEE semantics.

Floating-point operations in IEEE-754 are parameterized by a *rounding mode*, which specifies the value to return for inexact operations. Floating-point instructions in LLVM do not have a rounding-mode parameter,[13] and the language reference is inconsistent about whether LLVM makes any assumptions about rounding mode. If transformations must be correct under all rounding modes, then no transformation that may return zero can be correct, as the round-towards-positive and round-towards-negative modes require different results. Other parts of the reference state that transformations should assume rounding-towards-nearest, which is still ambiguous, because IEEE-754 provides two ways to break ties, towards even and away from zero. Alive does not specify a rounding mode, leaving the decision up to a particular encoding. By default, Alive-NJ uses rounding towards nearest, ties to even. Other rounding rules may be requested by the user.

Depending on the hardware, some operations that produce a NaN will trap, which is usually encoded as undefined behavior. Similarly, IEEE-754 includes a distinction between quiet and signalling NaNs. Instructions receiving an signalling NaN may trap. Some have argued that any division by zero in floating-point is undefined behavior, even though division of a non-zero value by zero is defined to return $\pm\infty$. The encoding described here assumes that operations returning NaN do not trap and assumes all NaNs are quiet NaNs.

3.3.4 Encoding instruction attributes

Instruction attributes modify instructions by promising that certain inputs will not occur. Adding **nsw** (no signed wrap) to an instruction allows the compiler to optimize

[13] Recent versions of LLVM simulate rounding-mode parameters with intrinsic functions that perform floating-point operations with a specific rounding mode.

Instruction	Poison-free condition
add nsw $\tau\, x, y$	$\rho_x \wedge \rho_y \wedge \mathsf{SExt}(\iota_x, 1) + \mathsf{SExt}(\iota_y, 1) = \mathsf{SExt}(\iota_x + \iota_y, 1)$
add nuw $\tau\, x, y$	$\rho_x \wedge \rho_y \wedge \mathsf{ZExt}(\iota_x, 1) + \mathsf{ZExt}(\iota_y, 1) = \mathsf{ZExt}(\iota_x + \iota_y, 1)$
sub nsw $\tau\, x, y$	$\rho_x \wedge \rho_y \wedge \mathsf{SExt}(\iota_x, 1) - \mathsf{SExt}(\iota_y, 1) = \mathsf{SExt}(\iota_x - \iota_y, 1)$
sub nuw $\tau\, x, y$	$\rho_x \wedge \rho_y \wedge \mathsf{ZExt}(\iota_x, 1) - \mathsf{ZExt}(\iota_y, 1) = \mathsf{ZExt}(\iota_x - \iota_y, 1)$
mul nsw int $w\, x, y$	$\rho_x \wedge \rho_y \wedge \mathsf{SExt}(\iota_x, w) \times \mathsf{SExt}(\iota_y, w) = \mathsf{SExt}(\iota_x \times \iota_y, w)$
mul nuw int $w\, x, y$	$\rho_x \wedge \rho_y \wedge \mathsf{ZExt}(\iota_x, w) \times \mathsf{ZExt}(\iota_y, w) = \mathsf{ZExt}(\iota_x \times \iota_y, w)$
udiv exact $\tau\, x, y$	$\rho_x \wedge (\iota_x \div_u \iota_y) \times \iota_y = \iota_x$
sdiv exact $\tau\, x, y$	$\rho_x \wedge (\iota_x \div \iota_y) \times \iota_y = \iota_x$
shl nsw $\tau\, x, y$	$\rho_x \wedge \rho_y \wedge (\iota_x << \iota_y) >> \iota_y = \iota_x$
shl nuw $\tau\, x, y$	$\rho_x \wedge \rho_y \wedge (\iota_x << \iota_y) >>_u \iota_y = \iota_x$
ashr exact $\tau\, x, y$	$\rho_x \wedge \rho_y \wedge (\iota_x >> \iota_y) << \iota_y = \iota_x$
lshr exact $\tau\, x, y$	$\rho_x \wedge \rho_y \wedge (\iota_x >>_u \iota_y) << \iota_y = \iota_x$

Table 3.11: Poison-free conditions for selected instruction attributes. The functions SExt and ZExt indicate sign-extension and zero-extension by a specified number of bits. The constant w is the width of the value in bits.

with the assumption that signed overflow will not occur (*e. g.*, it may assume that $x+1 > x$ always holds). Integer instructions return poison when the attribute conditions are not met. Floating-point instructions have undefined results if the "fast-math" attribute conditions are not met.

Addition-like integer arithmetic instructions may have one or both of the **nsw** and **nuw** (no unsigned wrap) attributes. Division-like instructions may have the **exact** attribute. The modified poison-free conditions for these instructions when an attribute is present are given in Table 3.11. When the **nsw** and **nuw** attributes occur on the same instruction, both poison-free conditions must hold.

The five floating-point arithmetic instructions and **fcmp** may take the "fast-math" attributes **nnan**, **ninf**, and **nsz**. These introduce a defined-result condition (see Section 3.2.5). If **nnan** (no NaN) is present, the condition is violated if the instruction's arguments or result are NaN. If **ninf** (no infinity) is present, the condition is violated if the arguments or result of the function are $\pm\infty$.

The **nsz** (no signed zeros) attribute introduces a controlled form of undefined behavior. The LLVM Language Reference states that optimizations may "treat the sign of a zero argument or result as insignificant" for instructions with the **nsz** attribute. We encode this by giving zero-valued arguments and results an `undef`-like sign bit.

For **nsz** instructions in the source, we are free to choose the signs for any zero-valued arguments or results in order to make the optimization correct. For **nsz** instructions in the target, the optimization must be correct for every possible choice of signs for zero-valued arguments.

For **fadd**, **fsub**, and **fmul**, differences in the signs of zero-valued arguments are only significant if the result is also zero-valued, so it is sufficient to make the sign of a zero-valued result `undef`:

$$\iota_z = \begin{cases} +0 & \text{if } \mathsf{isZero}(\iota_x \oplus \iota_y) \wedge b \\ -0 & \text{if } \mathsf{isZero}(\iota_x \oplus \iota_y) \wedge \neg b \\ \iota_x \oplus \iota_y & \text{if } \neg\mathsf{isZero}(\iota_x \oplus \iota_y) \end{cases} \tag{3.21}$$

$$\mathcal{Q}_z = \mathcal{Q}_x \cup \mathcal{Q}_y \cup \{b\} \tag{3.22}$$

where $\oplus$ encodes the floating-point operation, isZero is true for positive and negative zero, and b is a fresh Boolean variable.

The encoding of **fdiv** is similar, except that the sign of zero-valued arguments can also affect the sign of a non-zero result. Specifically, division by zero yields $\pm\infty$, depending in part on the sign of the zero. Thus:

$$\iota_z = \begin{cases} +0 & \text{if } \mathsf{isZero}(\iota_x) \wedge \neg\mathsf{isZero}(\iota_y) \wedge b \\ -0 & \text{if } \mathsf{isZero}(\iota_x) \wedge \neg\mathsf{isZero}(\iota_y) \wedge \neg b \\ +\infty & \text{if } \neg\mathsf{isZero}(\iota_x) \wedge \mathsf{isZero}(\iota_y) \wedge b \\ -\infty & \text{if } \neg\mathsf{isZero}(\iota_x) \wedge \mathsf{isZero}(\iota_y) \wedge \neg b \\ \iota_x \div \iota_y & \text{otherwise} \end{cases} \tag{3.23}$$

$$\mathcal{Q}_z = \mathcal{Q}_x \cup \mathcal{Q}_y \cup \{b\} \tag{3.24}$$

where b is a fresh Boolean variable.

3.3.5 Encoding comparison instructions

The **icmp** and **fcmp** instructions compare integer and floating-point values according to a specified relation.[14] LLVM provides ten ways to compare integers: **eq** and **ne** are

[14]LLVM frequently refers to this as a "predicate", but we will use "relation" to avoid confusion with Alive predicates.

		equal	unordered	unordered or equal
	false	**oeq**	**uno**	**ueq**
less than	**olt**	**ole**	**ult**	**ule**
greater than	**ogt**	**oge**	**ugt**	**uge**
less than or greater than	**one**	**ord**	**une**	**true**

Table 3.12: Organization of LLVM's floating-point relations. A relation is true if at least one of the column or row headers is true. For example, x **ult** y is true if $x < y$ or if x and y are unordered (*i. e.*, one or both is NaN).

equality and negated equality, **slt**, **sle**, **sge**, and **sgt** are signed inequalities, and **ult**, **ule**, **uge**, and **ult** are unsigned inequalities. These all correspond to relations available in the SMT bitvector logic.

The SMT relations return an SMT Boolean value, which must be converted to a 1-bit bitvector, which encodes LLVM's **i1** type. This is easily done using the SMT if construct, *e. g.*, $\mathsf{if}(\iota_x \bowtie \iota_y, 1, 0)$, where $\bowtie$ is the specific relation.

Floating-point comparisons are similar, but LLVM provides sixteen relations. These correspond to all possible disjunctions of four simpler relations: equal, less than, greater than, and *unordered*, where NaN values are not equal, less than, or greater than any value (including themselves) and are unordered with respect to all values. Table 3.12 shows how these four primitive relations can be combined into six ordered relations (which are always false if one or more argument is NaN), six unordered relations (which are always true if one or more argument is NaN), and four "leftover" relations. The ordered relations correspond to relations provided by the SMT floating-point logic, and the unordered relations can be found by negating the ordered relations (*e. g.*, **uge** is the negation of **olt**). The remaining four are trivial or can be encoded using isNaN.

3.3.6 Encoding select

The challenge of handling **select** comes not from the encoding itself, which is straightforward, but from the need to choose an interpretation first. Early versions of Alive used an "arithmetic-like" encoding, in which **select** is considered equivalent to a combination of bit-wise logical operators, as shown in Figure 3.8. Recent proposals [65]

```
  %r = select %c, %x, %y
=>
  %C = sext %c
  %N = xor %C, -1
  %a = and %x, %C
  %b = and %y, %N
  %r = or %a, %b
```

Figure 3.8: An arithmetic interpretation of **select**. The target is poison if either `%x` or `%y` is poison, regardless of which one is selected.

have favored a "branch-like" encoding, which interprets **select** as an implicit conditional branch followed by a choice of assignments. These interpretations coincide when the arguments to **select** are all poison-free, and also when one or more arguments has undefined behavior. They differ in how they behave in the presence of poison.

For an instruction **select** $c, \tau\, x, \tau\, y$, the differences between these interpretations have two aspects. First, if c, the *condition*, is poison, the result could be undefined behavior, poison, or a nondeterministic choice between x and y. Second, if one of x or y is poison, the result could always be poison (*unconditional poison*), or only be poison if the chosen value is poison (*conditional poison*).

These aspects may be decided independently, and Figure 3.9 shows five encodings resulting from these decisions.[15] The arithmetic-like encoding is given in Figure 3.9(c). Both Figure 3.9(b) and Figure 3.9(e) may be considered branch-like, depending on the intended behavior of conditional branches when the condition is poison.

The transformation in Figure 3.8 is incorrect for the interpretations with conditional poison. Its reverse is incorrect when a poison condition results in undefined behavior (as in Figures 3.9(a) and 3.9(b)).

Alive-NJ uses the arithmetic-like encoding by default, but the other four interpretations are provided to the user as options.

---

[15]The sixth possibility, combining nondeterministic choice and unconditional poison, is considered unreasonable.

Unconditional poison	Conditional poison
$\iota_z = \begin{cases} \iota_x & \text{if } \iota_c = 1 \\ \iota_y & \text{if } \iota_c \neq 1 \end{cases}$	$\iota_z = \begin{cases} \iota_x & \text{if } \iota_c = 1 \\ \iota_y & \text{if } \iota_c \neq 1 \end{cases}$
$\delta_z \equiv \delta_c \wedge \rho_c \wedge \delta_x \wedge \delta_y$	$\delta_z \equiv \delta_c \wedge \rho_c \wedge \delta_x \wedge \delta_y$
$\rho_z \equiv \rho_x \wedge \rho_y$	$\rho_z \equiv (\iota_c = 1 \wedge \rho_x) \vee (\iota_c \neq 1 \wedge \rho_y)$
(a)	(b) "Branch-like"
$\iota_z = \begin{cases} \iota_x & \text{if } \iota_c = 1 \\ \iota_y & \text{if } \iota_c \neq 1 \end{cases}$	$\iota_z = \begin{cases} \iota_x & \text{if } \iota_c = 1 \\ \iota_y & \text{if } \iota_c \neq 1 \end{cases}$
$\delta_z \equiv \delta_c \wedge \delta_x \wedge \delta_y$	$\delta_z \equiv \delta_c \wedge \delta_x \wedge \delta_y$
$\rho_z \equiv \rho_c \wedge \rho_x \wedge \rho_y$	$\rho_z \equiv (\rho_c \wedge \iota_c = 1 \wedge \rho_x) \vee (\rho_c \wedge \iota_c \neq 1 \wedge \rho_y)$
(c) "Arithmetic-like"	(d)
	$\iota_z = \begin{cases} \iota_x & \text{if } \rho_c \wedge \iota_c = 1 \\ \iota_y & \text{if } \rho_c \wedge \iota_c \neq 1 \\ \iota_x & \text{if } \neg\rho_c \wedge b \\ \iota_y & \text{if } \neg\rho_c \wedge \neg b \end{cases}$
	$\delta_z \equiv \delta_c \wedge \delta_x \wedge \delta_y$
	$\rho_z \equiv (\iota_c = 1 \wedge \rho_x) \vee (\iota_c \neq 1 \wedge \rho_y)$
	$\mathcal{Q}_z = \mathcal{Q}_c \cup \mathcal{Q}_x \cup \mathcal{Q}_y \cup \{b\}$
	(e) Nondeterministic choice

Figure 3.9: Five possible encodings of **select** $c, \tau\, x, \tau\, y$. The columns indicate whether poison from the non-selected value propagates. The rows indicate whether a poison condition causes undefined behavior, a poison result, or non-deterministic choice, respectively.

3.3.7 Encoding constant expressions

The semantics for Alive's constant expression sublanguage are based on LLVM's `APInt` type, which is used when translating Alive optimizations to C++ implementations. Most operations that correspond to a similar LLVM instruction have similar semantics, but `APInt` does not include `undef` or poison values, and has defined behavior in more circumstances than LLVM instructions.

Expressions in Alive's constant expression sublanguage may be: literal constants, symbolic constants, binary operators, or built-in functions. The encoding of symbolic constants is discussed in Section 3.3.2. Literals (*e. g.*, `1` or `-inf`) are encoded as SMT constants of the appropriate sort. Note that integer constants such as `1` are polymorphic over integer and floating-point types.

Binary operators

Integer expressions involving Alive's binary operators are encoded as SMT expressions using the corresponding function from SMT bitvector logic. The signed and unsigned division and remainder operators have special safety conditions that require their second argument be non-zero, in addition to requiring both arguments to be safe.[16]

Floating-point expressions are similarly encoded using the corresponding function from SMT floating-point logic, with the exception of **srem**, which is encoded using the fpMod function given in Figure 3.7. Floating-point operators are always defined, so no additional safety conditions are needed.

Functions

The built-in functions provided by Alive fall into several categories. Simple conversions (*e. g.*., **zext** or **fptrunc**) correspond to functions in SMT bitvector or floating-point logic. Alive always rounds towards zero when performing constant conversions involving floating point.

[16]Note that the safety condition for the constant operators **sdiv** and **srem** are different from the well-defined conditions for the corresponding instructions.

Function	χ
ComputeNumSignBits(x)	$\rho_x \implies b \leq_u \mathsf{signbits}(x)$
computeKnownOneBits(x)	$\rho_x \implies b \;\&\; \sim x = 0$
computeKnownZeroBits(x)	$\rho_x \implies b \;\&\; x = 0$

Table 3.13: Constant functions based on data-flow analyses

Other functions may have SMT encodings that are simple (*e. g.*, **max**(x, y) is encoded as $\mathsf{if}(x > y, x, y)$) or complex (*e. g.*, **log2** is encoded as a binary search for the most significant 1-bit). The functions **width** and **fpMantissaWidth** simply extract information from the argument's type, and are encoded as SMT constants.

Three additional functions correspond to analyses in LLVM, and are encoded as a free variable and an auxiliary condition given in Table 3.13. Note that the functions may return any value for poison arguments.

3.3.8 Encoding predicates

Predicates are encoded as SMT Boolean expressions. Negation, conjunction, and disjunction are encoded directly. Integer and floating-point comparisons are encoded using the corresponding SMT functions. Alive provides signed and unsigned comparisons for integer constant expressions, but provides only ordered comparisons for floating-point constant expressions. Note that equality and inequality use floating-point equality, where $0 = -0$ and NaN does not equal itself.

Alive's predicate functions can be divided into three groups, as in Table 3.14. The simplest take constant expressions as arguments and are encoded directly.[17] Functions that represent a dataflow analysis are encoded as a fresh Boolean variable and a side condition, as discussed in Section 3.2.6, except when the argument(s) are constant, in which case the condition always holds.

For example, **isPowerOf2**(a) is encoded as b and a side constraint $b \wedge \rho_a \implies \iota_a \;\&\; (\iota_a - 1) = 0 \wedge \iota_a \neq 0$, meaning that ι_a must be a power of two or poison when b holds.

[17] Many of these could be expressed as equalities, given some additions to Alive's constant function vocabulary.

Simple	Dataflow	Syntactic
fpIdentical	**CannotBeNegativeZero**	**hasNoInf**
fpInteger	**isPowerOf2**	**hasNoNaN**
isSignBit	**isPowerOf2OrZero**	**hasNSW**
isShiftedMask	**MaskedValueIsZero**	**hasNSZ**
WillNotOverflowSignedMul	**WillNotOverflowSignedAdd**	**hasNUW**
WillNotOverflowUnsignedMul	**WillNotOverflowUnsignedAdd**	**isConstant**
WillNotOverflowUnsignedShl	**WillNotOverflowSignedSub**	**isExact**
	WillNotOverflowUnsignedSub	**hasOneUse**

Table 3.14: Alive's built-in predicate functions, organized by type of analysis. All current dataflow analyses are must-analyses.

The third group are syntax tests and require their argument to be a value of a particular syntactic class (*e. g.*, a wrappable arithmetic instruction). These are encoded as true if the test is satisfied (*e. g.*, **isConstant**(c)). This can be useful for expressing profitability requirements. For example, an optimization that adds a **nsw** attribute to an instruction a may require !**hasNSW**(a) to avoid unnecessary work.

When the test is not trivially satisfied, the function is encoded as a unique fresh Boolean variable b and the encoding of the argument is modified to behave as though the syntactic condition were met when b is true. That is, **hasNSW**(a) modifies the encoding of a to behave as though the **nsw** attribute were present if b is true. Depending on the condition this may add poison-free conditions (as with **hasNSW**), potentially change the result (as with **hasNSZ**), or have no effect (as with **isConstant**).

Figure 3.10 shows an optimization using a combination of dataflow and syntactic analyses, and its SMT encoding.

3.4 Code Generation

In contrast to the SMT encoding, which interprets an Alive transformation as two abstract programs in order to compare their behavior, code generation interprets a transformation as a procedure for recognizing a pattern in an input program and replacing it. The Alive syntax is designed with this interpretation in mind. From the standpoint of verification, constant expressions in the source or variables not occurring in the source can easily be encoded into constraints, but allowing these would create ambiguity when generating code.

```
Pre: CannotBeNegativeZero(%x) \
       || hasNSZ(%r)
 %r = fsub %x, -0.0
=>
 %r = %x
```

(a)

$$\phi \equiv b_1 \vee b_2$$

$$\chi_p \equiv b_1 \implies x \neq -0$$

$$\iota_s = \begin{cases} +0 & \text{if } b_2 \wedge \mathsf{isZero}(x - -0) \wedge b_3 \\ -0 & \text{if } b_2 \wedge \mathsf{isZero}(x - -0) \wedge \neg b_3 \\ x - -0 & \text{otherwise} \end{cases}$$

$$\mathcal{Q}_s = \{b_3\}$$

$$\iota_t = x$$

(b)

Figure 3.10: An optimization involving dataflow and syntactic analyses. Note that $x - -0 = x$ except when $x = -0$. Thus, the optimization is only correct if `%x` is never -0 or if the sign of zero is irrelevant.

The code generated for an Alive transformation has the structure of an `if` statement, where the condition tests whether a particular instruction matches the source root (and, recursively, whether its arguments match the arguments of the source root) and whether the precondition is satisfied. If so, the body creates any new instructions defined by the target. The target root replaces the instruction matched by the source root. Any additional instructions are inserted into the program prior to the original instruction.

The core of InstCombine is a function resembling the schematic in Figure 3.11. The main loop of InstCombine attempts to transform each instruction by passing a pointer to a function like `optimize`, which looks for an applicable transformation. If no transformation applies, `optimize` returns a null pointer, indicating no change. If any transformation applies, `optimize` performs the changes made by the first one that applies and returns a value will replace the instruction being transformed.

The source pattern is implemented using LLVM's PatternMatch library, with care taken to correctly handle variables that are referenced more than once and to explicitly check type requirements when necessary. Target instructions are created using the LLVM API. Predicates are translated into function calls, usually to analysis functions already present in InstCombine, but occasionally to new functions written specifically to be used by Alive implementations. Figure 3.12 shows an implementation generated for the transformation given in Figure 1.2.

```
Value* optimize(Instruction *I)
{
  if (... match opt 1 ...) {
    ... create new target instructions ...
    return new_I;
  }
  if (... match opt 2 ...) {
    ... create new target instructions ...
    return new_I;
  }
  ...
  return null;
}
```

Figure 3.11: High-level structure of LLVM's InstCombine

```
ConstantInt *C2, *C1;
Value *m, *X;

if (match(I, m_SDiv(m_Value(m), m_ConstantInt(C2)))
      && match(m, m_NSWMul(m_Value(X), m_ConstantInt(C1)))
      && C2->getValue() != 0
      && C1->getValue().srem(C2->getValue()) == 0)
{
    BinaryOperator *r =
        BinaryOperator::CreateMul(X,
            ConstantInt::get(I->getType(),
                C1->getValue().sdiv(C2->getValue())));
    r->setHasNoSignedWrap(true);
    return r;
}
```

Figure 3.12: C++ code generated for the transformation in Figure 1.2

```
APInt InstCombiner::computeKnownOneBits(Value *V) {
  unsigned BitWidth = V->getType()->getScalarSizeInBits();
  APInt KnownZero(BitWidth, 0), KnownOne(BitWidth, 0);
  computeKnownBits(V, KnownZero, KnownOne);
  return KnownOne;
}
```

Figure 3.13: A wrapper function used by the translation of `computeKnownOneBits`. InstCombine's `computeKnownBits` modifies its arguments rather than returning a result, so it cannot be used directly by the translation scheme employed by the code generator.

Integer constant expressions translate to computations in LLVM's `APInt` type. This limits Alive somewhat, as LLVM constant expressions may also include `undef` values and link-time values. However, LLVM constant expressions do not support certain operations, such as comparisons. It would be possible to use a hybrid translation, using LLVM constant expressions whenever possible and restricting to `APInt` when necessary, but using `APInt` throughout is simpler and more predictable.

The binary operators for constants correspond to methods provided by `APInt`. Constant functions correspond to C++ functions provided by LLVM or specially written for Alive. For example, InstCombine's `computeKnownBits` is used to translate `computeKnownOneBits`, but its signature does not allow it to be used directly by the translation. Instead, `computeKnownOneBits` is translated to a wrapper function shown in Figure 3.13. Certain C++ binary operators and functions are overloaded, so that specific arguments may be an `APInt` or a C++ integer. The code generator takes advantage of that when it can, as with the comparisons with zero in Figure 3.12.

Floating-point constant expressions would analogously translate to computations in LLVM's `APFloat` type, but its operations do not support the coding style currently used by the code generator. Demand for floating-point constant expressions has not yet been sufficient to justify the necessary re-engineering of the code generator.

3.4.1 Type references

The implementation of a transformation may need to refer to type information in some places. For example, creating a conversion instruction or a literal constant require explicit type information. As Alive transformations are polymorphic, it is usually not

```
Pre: WillNotOverflowUnsignedAdd(%a, %b)
  %x = zext %a
  %y = zext %b
  %z = add %x, %y
=>
  %c = add %a, %b
  %z = zext %c
```

Figure 3.14: A transformation with an implicit type constraint. In order for the precondition and target to be well-typed, `%a` and `%b` must have the same type, but this type equality is not implied by the source.

possible to give a specific, concrete type. Instead, the code generator associates each type variable with a value from the source that is constrained to have the same type. References to the type variable can then be filled by using the `getType` method of the associated source value.

Because of this requirement, all values in the precondition and target must be equal in type to some value in the source. The exception are comparisons in the precondition, which have a default type if they are not otherwise constrained. This permits comparisons such as `width(%a) u> 1`, which would otherwise be ambiguous.

InstCombine operates on well-typed LLVM IR, so an Alive transformation is free to assume that the code matched by the source pattern satisfies its type constraints. This is usually sufficient to ensure that the transformation will not create ill-typed code, but not always. The target or precondition may introduce type constraints that are not already required by the source, as in Figure 3.14. The code generator uses an augmented type checker, which detects when the target or precondition equate two types that were not already equal, and adds explicit type equality checks as needed.

3.4.2 Matching the source

The translation of the source primarily uses LLVM's PatternMatch facility, which tests whether a value meets some criteria and optionally binds arguments for further matching. For example, consider this pattern match:

```
match(I, m_NSWAdd(m_Value(a), m_ConstantInt(C1)))
```

The call to `match` tests whether the instruction `I` is an `add` with `nsw`, where the first

argument is an arbitrary value and the second is an integer constant. If the test succeeds, then `a` and `C1` will be bound to the first and second argument, respectively, as a side-effect of the match. This is similar to pattern matching in languages that provide algebraic data types.

The source of an Alive transformation contains instructions, input variables, and literal and symbolic constants. For the most part, these correspond to patterns provided by LLVM. Some combinations of features are not directly provided, such as matching an instruction with more than one attribute, but can be built by combining other features. The patterns for matching `icmp` and `fcmp` do not specify the condition. Instead, the condition is bound to a new variable, which must be checked separately.

Care must be taken when a value is referenced multiple times in the source. LLVM provides a pattern `m_Specific(`x`)`, which succeeds if the value being matched is equal to x.[18] If two instructions refer to the same value, the pattern for the first instruction[19] will bind the value, and the second can use `m_Specific`. However, `m_Specific` cannot be used if a single instruction refers to a previously-unbound value twice. Instead, both references are bound to different variables, and their equality is checked separately.

For example, the source `select %a, %a, %b` becomes the pattern match

```
match(I, m_Select(m_Value(a), m_Value(a_0), m_Value(b))) && a == a_0
```

3.4.3 Testing the precondition

Predicates are Boolean-valued, so they can be used in the `if` condition directly. Conjunction, disjunction, and negation are translated to the corresponding C++ constructs. Integer comparisons translate to comparisons of `APInt` values. As C++ does not provide separate signed and unsigned comparisons, the translations use methods provided by `APInt`.

---

[18] Note that equality for LLVM instructions is pointer equality. Two instructions with the same opcode and arguments may or may not be considered equal.

[19] That is, the instruction whose pattern is matched first, which may not be the instruction that is defined first in the transformation. Recall that pattern matching begins with the root instruction and then proceeds to its arguments.

Predicate functions are translated to function calls. Most such functions are already present in InstCombine, but a few are specially written to support Alive-generated code, *e. g.*. `WillNotOverflowSignedMul`. The InstCombine-provided functions occasionally require additional context arguments, but these can be provided in a standard way.

3.4.4 Creating the target

The target of a transformation creates zero or more instructions, and assigns a value to replace the source root. Instructions are created using the appropriate constructor or factory function. If more than one instruction is created, all instructions but the last are passed to InstCombine's instruction builder, which inserts the new instruction into an appropriate basic block. The final instruction, or other value replacing the root, is returned to the InstCombine's main loop, which replaces all references to the original root instruction with the new value.

This asymmetry between the treatment of the last instruction and those preceding it is done to maintain compatibility with InstCombine.

3.5 Extensions to Alive

This chapter primarily discusses the dialect of Alive supported by the Alive-NJ toolkit. The original Alive toolkit [75] does not support floating-point types, but has several other features that are not present or incompletely supported by Alive-NJ.

3.5.1 Relation variables

The grammar in Section 3.1.1 requires the relation used by **icmp** and **fcmp** to be one of the relations included in *rel* and *frel*, respectively. That is, each transformation involving a comparison instruction must require a specific relation be tested. Some transformations involving comparisons are specific examples of a more general transformation that is generic over relations. If the relation can be made abstract, these transformations can be combined.

```
Pre: fpext(C2) == C1
  %x = fpext %a
  %r = fcmp %x, C1
=>
  C2 = fptrunc(C1)
  %r = fcmp %a, C2
```

Figure 3.15: A transformation using an implicit relation variable

The abstract syntax is augmented by adding a new class of variable names, representing relation variables, and allowing them to occur in *rel* and *frel*, with a restriction to prevent variables from occurring in comparison predicates and a requirement that any relation variable occurring in the target be defined in the source. A relation variable will be encoded as a bit vector of width four (enough to uniquely represent its possible values), and comparison instructions using that variable will select which relation to use based on its value. The code generator must check that any repeated use of a relation variable in the source matches the same relation, and to capture relation variables for use in the target.

For convenience, a blank relation variable in the concrete syntax is interpreted as a variable whose name is derived from the name of its instruction. This is only useful when the comparison is the root (except in dialects with multiple replacement, see Section 3.5.3).

For example, the transformation in Figure 3.15 generalizes the transformation in Figure 3.4 to apply for all relations.

Alive-NJ supports predicate variables, but they are not widely used.

Relation functions

Even more transformations can be generalized if the language provides functions on relations or relation equality predicates. For example, `reverse(rel)` might reverse the direction of the relation `rel`, which could be used in the transformation in Figure 3.16(a), which negates both sides of a comparison. In the Alive-NJ dialect, this must be expressed as ten transformations.

```
  %a = sub 0, %x
  %b = icmp rel %a, C
=>
  %b = icmp reverse(rel) %x, -C
```

(a)

```
Pre: rel == ord || rel == oeq \
    || rel == oge || rel == ole
  %f = fcmp rel %x, %x
=>
  %f = fcmp ord %x, 0
```

(b)

Figure 3.16: Two transformations using a relation function and a relation predicate, respectively

Similarly, if relations can be used in equality comparisons, this allows generalizing over subsets of possible relations. Figure 3.16(b) shows one such example, which generalizes four transformations.

Naturally, it would need to be determined if the gain in expressivity is sufficient to justify the added complexity to the language.

3.5.2 Named type variables

In the Alive-NJ dialect, types for instructions may be given explicitly or omitted. Other dialects also allow explicit type variables. If a type variable occurs in multiple places, those places are constrained to have the same type. For example, named type variables in Figure 3.14 could make the implicit constraint that `%a` and `%b` have the same type into an explicit requirement.

Except when making implicit requirements explicit, the use of type variables can only restrict the feasible type assignments for a transformation, making it less applicable than it might otherwise be. It is unclear whether a transformation exists that requires such a restriction to be correct.

However, combined with type annotations for constant expressions, some expressions that currently require bound symbolic constants may be specified without them. For example, Figure 3.17 shows the transformation in Figure 3.4 rewritten to use a type variable, `ty1`, and an annotation in the expression `fptrunc(C1) : ty1` that fixes its type, preventing it from being ambiguous and ensuring that both truncations of `C1` truncate to the same type. (The version using a bound symbolic constant still has the advantage of only truncating `C1` once.)

```
Pre: fpext(fptrunc(C1) : ty1) == C1
  %x = fpext %a to ty1
  %r = fcmp oeq %x, C1
=>
  %r = fcmp oeq %a, fptrunc(C1)
```

Figure 3.17: A transformation using a named type variable and a type annotation

```
Pre: isPowerOf2(%A) && hasOneUse(%Y)
  %Y = lshr i8 %A, %B
  %r = udiv %X, %Y
=>
  %Y = lshr exact %A, %B
  %r = udiv %X, %Y
```

Figure 3.18: A multiple-replacement transformation. Both instructions will be replaced when the transformation is applied.

3.5.3 Multiple replacement

As described in Section 3.1.1, the roots of the source and target must have the same name, because the target root will replace the source root when the transformation is applied. Any other instructions in the target must not have the same name as a source instruction, and no other instructions in the source will be replaced. This may be called a *single-replacement* transformation.

A dialect of Alive may generalize this by allowing *multiple-replacement* transformations, in which several instructions in the target have the same name as instructions in the source and replace them when the transformation is applied. Figure 3.18 shows a transformation derived from LLVM that uses multiple replacement.

Each replaced instruction may be referenced by other instructions outside the source pattern, so it is necessary to perform the correctness checks for each replaced instruction, not just the roots.

However, consider the example in Figure 3.18 more carefully. Adding the **exact** attribute to `%Y` is acceptable because a situation where `%Y` would become poison after transformation is one where it would have returned 0 before translation, causing undefined behavior in `%r`. If `%Y` were used by multiple instructions, and `%r` were only evaluated in cases where `%Y` is non-zero, this transformation would be invalid because

it introduces poison. Thus, the requirement that `%Y` has only one use is essential—but in that case, replacing it is no different from changing `%r` to refer to a new **lshr exact** instruction and letting `%Y` be eliminated as dead code.

When replacing a non-root instruction that may have uses outside the source pattern, the verifier can't take advantage of any contextual information provided by its context (such as the requirement that `%Y` be non-zero). In such a case, its replacement could just as easily be done by a different transformation.

In short, it is not clear that multiple replacement adds any expressive power to Alive. In particular, all transformations derived from LLVM using multiple replacement were expressible using single replacement.

3.5.4 Memory operations

The dialect of Alive supported by the Alive-NJ toolkit describes computations over values stored in temporary registers. Including memory-related instructions, as in the initial presentation of Alive [76], increases the number of transformations that can be specified, but requires the correctness conditions to incorporate a memory model, in addition to the extensions to the syntax and type system.

Basic support for memory operations requires the addition of pointer types to the type system and the instructions **load** and **store**, which read from and write to memory, respectively. The conversions **inttoptr** and **ptrtoint** can be added to allow unstructured memory access and pointer arithmetic. The **alloca** instruction allocates new memory that does not overlap any existing memory blocks.

Structured memory access requires support for aggregate types, such as arrays, and the instruction **getelementpointer**, which finds an offset into an aggregate type using one or more indices.

The pointer aliasing rules for LLVM require that all pointers used to access memory to be associated with an address range, or else the access has undefined behavior. A pointer that is based on another, meaning its value was computed or otherwise derived from that pointer, is associated with the same address range. The address ranges allocated by **alloca** do not overlap with any other address range. In Alive, nothing is

$$
\begin{array}{rcl}
\mathit{stmt} & ::= & \cdots \mid \textbf{store}\ \mathit{type}\ \mathit{val}, \mathit{type}\ \mathit{val} \\
\mathit{inst} & ::= & \cdots \mid \textbf{alloca}\ \mathit{type}, \mathit{const} \mid \textbf{load}\ \mathit{type}, \mathit{type}\ \mathit{val}, \textbf{align}\ \mathit{width} \\
 & & \textbf{getelementptr}\ \mathit{type}, \mathit{type}\ \mathit{val}, \overline{\mathit{type}\ \mathit{val}} \\
\mathit{type} & ::= & \cdots \mid \textbf{ptr}\ \mathit{type} \mid \textbf{array}\ \mathit{const}\ \mathit{type} \\
\mathit{conv} & ::= & \cdots \mid \textbf{inttoptr} \mid \textbf{ptrtoint} \\
\mathit{const} & ::= & \cdots \mid \textbf{null}
\end{array}
$$

Figure 3.19: Extended abstract syntax for memory operations

known about pointers not based on an **alloca**, so they must be considered to potentially overlap.

The addition of memory accesses and instructions that do not return a value (*i. e.*, **store**) creates additional requirements for determining which programs match the source pattern. While non-memory instructions may be equivalently performed in any order, as long as each instruction is computed after its arguments, memory operations do not generally commute. If the source pattern include memory accesses, these accesses must form a chain of memory states in the input program and cannot be interleaved with other memory accesses not matched by the source. (This rule can be relaxed somewhat for non-volatile memory accesses and accesses to memory known not to alias with the memory accessed by the transformation.) Store instructions do not have names, so they must be present in the source and target. This, in turn, requires that the target not reuse the names of load instructions, as their position in the chain of memory accesses is unclear. Thus, transformations involving **store** will most likely require multiple replacement (see Section 3.5.3).

Additions to syntax and type system

Figure 3.19 shows the additions to the syntax needed for basic and structured memory access. The **ptr** type will be encoded as a 32- or 64-bit pointer. The **array** type represents one of LLVM's several aggregate types. Note that the size of an array is a constant expression. Alive does not specify whether the symbolic constants used in an array type can also be used as values. The arguments to **load** are the pointer to read from and an optional alignment. The arguments to **store** are the value to be written and the destination pointer.

$$\frac{\Gamma \vdash x : \mathbf{int}\, 1 \quad \Gamma \vdash y : \tau \quad \Gamma \vdash z : \tau \quad \tau \in I \cup F \cup P}{\Gamma \vdash \mathbf{select}\, x, \tau\, y, \tau\, z : \tau}\ \text{SELECT}$$

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash y : \tau \quad \tau \in I \cup P}{\Gamma \vdash \mathbf{icmp}\, \tau\, x, y : \mathbf{int}\, 1}\ \text{ICMP} \qquad \frac{\tau \in P}{\Gamma \vdash \mathbf{null} : \tau}\ \text{NULL}$$

$$\frac{\Gamma \vdash c : \tau' \quad \tau' \in I}{\Gamma \vdash \mathbf{alloca}\, \tau, \tau'\, c, \mathbf{align}\, n : \mathbf{ptr}\, \tau}\ \text{ALLOCA}$$

$$\frac{\Gamma \vdash x : \mathbf{ptr}\, \tau \quad \tau \in I \cup F \cup P}{\Gamma \vdash \mathbf{load}\, \mathbf{ptr}\, \tau\, x : \tau}\ \text{LOAD} \qquad \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash y : \mathbf{ptr}\, \tau}{\Gamma \vdash \mathbf{store}\, \tau\, x, \mathbf{ptr}\, \tau\, y\ \text{ok}}\ \text{STORE}$$

$$\frac{\Gamma \vdash x : \mathbf{ptr}\, \tau' \quad \tau' \in P \quad \overline{\Gamma \vdash y_i : \tau_i} \quad \overline{\tau_i \in I} \quad \tau' \xrightarrow{\overline{y_i}} \tau}{\Gamma \vdash \mathbf{getelementpointer}\, \tau, \mathbf{ptr}\, \tau'\, x, \overline{\tau_i\, y_i} : \tau}\ \text{GEP}$$

Table 3.15: Extended typing rules for memory operations

conv	$\tau_1 \overset{conv}{\bowtie} \tau_2$
bitcast	$\tau_1 =^w \tau_2 \quad \tau_1 \in I \cup F \quad \tau_2 \in I \cup F$
bitcast	$\tau_1 \in P \quad \tau_2 \in P$
inttoptr	$\tau_1 \in I \quad \tau_2 \in P$
ptrtoint	$\tau_1 \in P \quad \tau_2 \in I$

Table 3.16: Extended conversion relations for memory operations

Tables 3.15 and 3.16 describe the additions to the typing rules required for memory operations. The set P includes all possible pointer types. Note that **icmp** may be used to compare pointers, and that **select** is restricted to types in $I \cup F \cup P$, the types of *first-class values*. The **bitcast** instruction is now limited to convert between integer and floating-point values of the same width, or between arbitrary pointers.[20] Only **inttoptr** and **ptrtoint** can convert between pointer and non-pointer types.

The **getelementpointer** instruction is unusual in that it takes an arbitrary number of arguments. Excluding types, the first argument will be a pointer to an aggregate type and subsequent arguments will be integer indices into its nested aggregate. The relation $\tau \xrightarrow{x} \tau'$ indicates that the element in a τ aggregate at index x has type τ'. Similarly, $\tau \xrightarrow{\overline{x_i}} \tau'$ indicates a sequence of lookups into a nested structure. In a dialect of Alive supporting only **array** aggregates, each element will have the same type and a **getelementpointer** with $n+1$ arguments will index into an n-dimensional array.[21]

[20] Alive does not support multiple address spaces, so all pointers width-equal.

[21] Structures in LLVM may have elements with different types, but their indices must be constant to allow for type checking. For Alive, this would mean that the return type will constrain the possible

The concrete syntax for **array** τ c is `[`τ `x` c`]` and for **ptr** τ is τ`*`. A pointer to an array of c_1 8-bit integers is `[i8 x C1]*`. The arguments to **getelementpointer** are separated by commas. Both arguments to **alloca** are optional, with the number of elements to allocate defaulting to one. All arguments to **load** except the pointer are optional; the first comma is omitted if the result type is defaulted and the second comma and **align** keyword are omitted if the alignment is defaulted. The default alignment is determined by the result type and the ABI in use.

The DAG representation can be extended to work with memory operations by adding a new type of edge representing the memory chain. Each **load**, **store**, and **alloca** node will have exactly one incoming and out outgoing memory edge, with special nodes indicating the initial and final memory states for the source and target. Alternatively, to reflect the commutativity of **load** operations, each **load** would have a single incoming edge, and a **store** might have several outgoing edges, all but one leading to a **load**.

Encoding memory operations

Memory is represented as an array of bytes indexed by 32- or 64-bit integers. The source and target begin with an initial memory array m_0 and produce final memory arrays m_s and m_t, respectively. Operations on this array can be encoded using the SMT theory of arrays, or by using Ackermann's expansion [4]. The well-defined conditions for any memory operation include the requirement that all prior memory operations were well-defined.

Memory accesses in LLVM must be done through a pointer associated with an address range, or else that access has undefined behavior. In an Alive transformation, all pointers come from input variables or **alloca** instructions, or are *based on* such a pointer through conversions and **getelementpointer** operations. In particular, converting a pointer to an integer, performing some arithmetic, and then converting back to a pointer creates a new pointer that is based on the original. The based-on relation is transitive.

---

values the index could take.

Each **alloca** instruction allocates an address range that does not overlap any other memory range. The pointer returned by **alloca** is constrained (1) to not be **null**, (2) to be properly aligned, (3) to not overlap with any other allocated address range, and (4) to not wrap around the memory space. Note that these rules are guaranteed by the semantics of the instruction, not promises made about its arguments. Thus, they would be encoded as auxiliary conditions (χ), not well-defined conditions (δ). The newly-allocated address range is uninitialized, similar to an `undef` value. One way to represent this property is to allocate a fresh bit-vector variable as large as the address range, write its contents to memory, and then add the variable to $\mathcal{Q}$. This ensures that any bits left uninitialized by the source will not be considered changed if they are written to by the target, but is insufficient to fully capture the behavior of `undef`, since multiple reads from the same initialized location will always return the same bit pattern.

The **load** instruction is implemented by reading multiple bytes from the memory array and concatenating the results into the return value. It is undefined behavior to read from a null pointer or from a location not in any address range or if the pointer is poison.

The **store** instruction is implemented by dividing the stored value into bytes and storing each byte into memory individually. No memory is written if undefined behavior has occurred prior to this instruction. It is undefined behavior to write to a null pointer or from a location not in any address range or if the pointer or value is poison.

Pointers derived from input variables are considered to be part of an unknown address range. Nothing can be known about these address ranges, except that they do not include any memory allocated by **alloca** or (out-of-range) accessed by a pointer based on a pointer returned by **alloca**.

This encoding promotes reads and write of poison to undefined behavior. An alternative encoding tracks each byte for poison status using a second array. Reading a value returns poison if any of its bytes are poison, and storing poison marks all bytes written to as poison.

Memory correctness condition

The constraints for **alloca** are stored in auxiliary conditions, which necessitates adding auxiliary conditions for the source (χ_s) to the last three correctness conditions in Section 3.2.7. A sixth correctness condition then requires the final states of memory to be equal:

$$\forall_{\mathcal{V},i}\, \exists_{\mathcal{Q}_s} : \chi_s \wedge \chi_t \wedge \chi_p \wedge \phi \implies \mathsf{select}(m_s, i) = \mathsf{select}(m_t, i) \tag{3.25}$$

where i ranges over addresses and $\mathsf{select}(m, i)$ looks up index i in array m. Note that this equality is required to hold even if the source has undefined behavior or returns poison. In LLVM, undefined behavior does not propagate backward, so it cannot undo prior memory operations. For this reason, **store** and **alloca** operations have no effect if they occur after an instruction with undefined behavior.

On the other hand, one could argue that a program with undefined behavior has no defined final memory state, because its possible behaviors include all possible combination of **store** instructions. Under this interpretation, the correctness check for memory only applies when the source is well-defined:

$$\forall_{\mathcal{V},i}\, \exists_{\mathcal{Q}_s} : \chi_s \wedge \chi_t \wedge \chi_p \wedge \phi \wedge \delta_s \implies \mathsf{select}(m_s, i) = \mathsf{select}(m_t, i). \tag{3.26}$$

The memory states must still be equal if the source returns poison, because it is possible that the poison value will not cause undefined behavior.

An argument could be made for an additional correctness condition that ensures any two pointers which refer to distinct address ranges in the source also refer to distinct address ranges in the target. Consider a statement `%r = select %c, %p, %q` that chooses between two pointers. This statement implies that `%r` is based on `%p` or `%q` but does not imply any relationship between `%p` and `%q`. A transformation might replace the **select** instruction with an arithmetic expression, as seen in Figure 3.8, after first converting `%p` and `%q` to integers and the converting the result back to a pointer. Ignoring poison values, such a transformation does not change the computed result or any memory accesses, but in the target `%r` is now based on both `%p` and `%q`, implying a relationship between them due to transitivity. If `%p` and `%q` were both based on separate

alloca instructions, this transformation has now caused any memory access using `%r` to be undefined.

3.5.5 Combining poison and `undef`

Recent work [65] has proposed changing the semantics of LLVM to combine the ideas of `undef` and poison values. Operations that previously returned an `undef`-like value or undefined results (see Section 3.2.5) instead return poison values.

A **freeze** instruction is added that has no effect on non-poison values, but replaces poison values with an arbitrary bit pattern. Like `undef`, different **freeze** instructions applied to the same poison value will get different bit patterns. Unlike `undef`, different references to the same **freeze** instruction will get the same bit pattern. Freezing poison values is necessary to prevent undefined behavior in some circumstances. For example, a conditional branch has undefined behavior if the condition is poison, as opposed to non-deterministic behavior if the condition is `undef` or frozen poison. The encoding of **freeze** $\tau\, x$ is

$$\iota = \begin{cases} \iota_x & \text{if } \rho_x \\ u & \text{if } \neg\rho_x \end{cases} \tag{3.27}$$

$$\mathcal{Q} = \mathcal{Q}_x \cup \{u\} \tag{3.28}$$

$$\rho \equiv \top \tag{3.29}$$

where u is a fresh variable consistently chosen for this instruction.

Alive-NJ includes an encoding using the proposed semantics. The **freeze** instruction can be used with any encoding, but by default it is encoded as a no-op.

3.6 The Alive-NJ Toolkit

The Alive-NJ toolkit includes a verifier for Alive transformations using the dialect of Alive described in Section 3.1 according to the interpretation in Sections 3.2 and 3.3. It was originally created as part of the Alive-FP project for verifying transformations involving floating-point operations [87], and is a ground-up rewrite of the original Alive toolkit [75]. Based on the experience of creating the original toolkit and the Alive-Loops

termination checker [85] (see Chapter 5), Alive-NJ is designed to facilitate experiments with extensions to Alive and alternative semantics, and as a foundation for building additional tools, such as the Alive-Infer precondition inference tool [86] described in Chapter 4. The implementation is roughly 6,200 lines of Python, with an additional 2,000 lines for Alive-Infer. SMT queries are resolved using the Z3 solver [28].

Internally, Alive-NJ uses the DAG representation of Alive transformations, with each instruction, input variable, constant expression, and predicate represented as an object. A transformation includes references to the roots of the source, target, and precondition. Alive-NJ additionally allows transformations to have an assumption, which is a special precondition used for precondition inference (see Section 4.3.2). Assumptions are used by verification, but are not checked by the generated implementation. The representation is treated as immutable: Alive-NJ never modifies a representation object after it is created.

Type checking is performed by collecting constraints for each term in a transformation. Terms that must have the same type are unified, and other requirements such as width-ordering or -equality are captured separately. Type checking succeeds if the constraints contain no contradictions and produces a *type environment*. This environment maps each term to a type variable, and includes the type-correctness constraints expressed in terms of these variables. Using a type environment, one can then enumerate *type models*, which assign a concrete type to each type variable.

Separating the type environment from the DAG representation allows terms to be reused in different contexts, which can be convenient for rapid prototyping. Separating the abstract type environment from the concrete type model simplifies creation of correctly typed, type-parametric terms (for example, by the predicate enumerator described in Section 4.4.4).

Alive-NJ is designed to support multiple encodings of Alive terms, corresponding to different interpretations of the semantics or even different methods of correctness checking. Encoding is performed by an encoder object, which produces the six parts of the SMT encoding for a term with respect to a type model. New encodings can be added by creating a subclass of `BaseSMTEncoder` and overriding its behavior where

```
import alive.language as L
import alive.smtinterp as E
import alive.tools.verify as V
import z3

class RotlInst(L.IntBinaryOperator):
    code = "rotl"

@E.eval.register(RotlInst, E.BaseSMTEncoder)
def _(term, encoder):
    x = encoder.eval(term.x)
    y = encoder.eval(term.y)
    ty = encoder.type(term)

    y = z3.URem(y, ty.width)

    return (x << y) | z3.LShR(x, ty.width - y)

if __name__ == '__main__':
    V.main()
```

Figure 3.20: A Python module that adds a `rotl` (rotate left) instruction to Alive. The `eval` method dispatches based on the class of the term and the encoder being used and returns an SMT expression encoding its result. Well-defined and poison-free conditions propagate automatically.

desired. Alive-NJ includes subclasses that encode undefined results as `undef` or poison (see Section 3.2.5), all five interpretations of **select** (see Section 3.3.6), and the proposed unification of `undef` and poison (see Section 3.5.5).

In addition to different encodings, Alive-NJ is designed to enable experimentation with additions to the language. In particular, new binary arithmetic and conversion instructions, constant functions, and predicate functions can be added to the language without modifying any code in Alive-NJ itself. Figure 3.20 shows a complete Python module that extends Alive with a `rotl` (rotate left) instruction. This module may be imported by other tools, and may be executed as a script to verify transformations in its extended Alive language.

The module has three parts. First, it declares the class `RotlInst`, which represents a rotate left instruction internally. Alive-NJ uses metaprogramming facilities in Python so that declaring this class automatically extends the grammar of the binary operator instructions to include `rotl`. This affects both the parser and the pretty-printer, so

importing this module gives Alive-NJ the ability to read and write Alive transformations that use `rotl`. Because `RotlInst` inherits from `L.IntBinaryOperator`, it will constrain its arguments and result to have the same integer type.

The next block of code extends the `eval` function to be able to encode `rotl` instructions. Alive-NJ implements `eval` using double-dispatch, so that its behavior can be specialized for specific classes of terms and for specific encoders. In this case, we are registering a method for `RotlInst` and the base encoder class. The body of the method uses the encoder to evaluate the arguments of the instruction and its return type. Next, it finds the remainder of `y` divided by the bit width of its return type, because rotating a w-bit integer by $nw + y$ places is equivalent to rotating by y places. Finally, it computes the result by combining a left shift by y places with a logical right shift by $w - y$ places.

The final part of the code is executed if the module is executed as a script, as opposed to importing it. Once the new instruction has been created and inserted into the parser, pretty-printer, and encoders, it invokes the built-in verification tool, which will interpret command-line options, read files, and verify transformations as usual, but now with support for an additional instruction.

3.7 Evaluation

During the development of Alive and the Alive toolkit, we created several hundred Alive transformations by translating code present in the InstCombine and InstructionSimplify phases of LLVM. These transformations were checked for correctness and used to generate a new peephole optimizer, which was tested for performance and for the performance of the code it optimized.

The goals of Alive go beyond checking the correctness of existing code, useful as that may be. Alive is intended to simplify the development of new peephole optimizations, and to make it less likely that incorrect transformations will be added to LLVM. Alive has detected errors in proposed transformations, and is now commonly used to check the correctness of proposals.

By providing a framework for automated correctness checking and a large collection of transformations that are known to be desirable, Alive has also proven useful for testing proposed changes to the LLVM semantics. The effect of proposals can be measured by creating a modified Alive toolkit (or adding a new encoding to Alive-NJ) that uses the new semantics and then observing whether any transformations change correctness status.

3.7.1 Translation of transformations from LLVM

Concurrent with the development of Alive and subsequent extensions, several hundred transformations implemented by the InstructionSimplify and InstCombine passes in LLVM have been translated into Alive specifications. The purposes for this were twofold: to provide a body of real-world transformations specified in Alive, for the purposes of better understanding the language and judging its implementation, and to assess the correctness of transformations already performed by Alive. During the initial development, 334 transformations were translated to Alive. Alive-NJ currently includes a suite of 417 integer-based transformations, 96 floating-point–related transformations.

Of the original 334 transformations, eight (2.4%) were found to be incorrect (see Figure 3.21). The most common form of bug was the introduction of undefined behavior for some inputs that were defined for the source pattern, with the remainder introducing poison or changing the computed result. Subsequent work has discovered six integer-related bugs and nine floating-point–related bugs, six found by Alive-FP [87] (see Figure 3.22) and three found by others [95], for a total of 23 bugs in existing code discovered as of this writing.

The time needed to verify a transformation varies greatly. Verifying a transformation requires solving up to five SMT queries for each type assignment. Even when type assignments are limited to those not exceeding 64-bit integers, a transformation involving conversion instructions can easily have several thousand type assignments. Most queries are resolved in a fraction of a second, but queries that depend on non-linear arithmetic, particularly multiplication and division, will take increasingly long as the

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

(a)

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

(b)

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

(c)

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

(d)

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

(e)

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

(f)

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

(g)

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

(h)

Figure 3.21: Incorrect integer arithmetic transformations found in LLVM

```
Name: PR26746
Pre: C == 0.0
 %1 = fsub -0.0, %x
 %r = fsub C, %1
=>
 %r = %x
```

(a)

```
Name: PR27151
Pre: C == 0.0
 %y = fsub nnan ninf C, %x
 %z = fadd %y, %x
=>
 %z = 0
```

(b)

```
Name: PR26862-1
 %r = fdiv undef, %x
=>
 %r = undef
```

(c)

```
Name: PR26862-2
 %r = fdiv %x, undef
=>
 %r = undef
```

(d)

```
Name: PR26863-1
 %r = frem undef, %x
=>
 %r = undef
```

(e)

```
Name: PR26863-2
 %r = frem %x, undef
=>
 %r = undef
```

(f)

```
Name: PR27153
Pre: sitofp(C2) == C1 && WillNotOverflowSignedAdd(%a, C2)
  %x = sitofp %a
  %r = fadd %x, C1
=>
  C2 = fptosi(C1)
  %y = add nsw %a, C2
  %r = sitofp %y
```

(g)

Figure 3.22: Incorrect floating-point transformations found in LLVM

size of the integers increases. To avoid spending several hours on verification, transformations in the Alive suite involving non-linear arithmetic are often restricted to use a single type assignment involving small integers.

3.7.2 Performance of generated implementation

A new peephole optimization pass was created by generating implementations for the 334 transformations originally derived from LLVM. The performance of this implementation was compared against LLVM 3.6 by comparing the time needed to compile LLVM's test suite and the SPEC 2000 and SPEC 2006 benchmarks, and by comparing the performance of the programs.

The new peephole optimizer uses the infrastructure of InstCombine, but replaces the call to `visit`, which attempts to transform a single instruction, with a call to a new function created by the code generator. This function tested the instruction on every transformation that had a root instruction with the same opcode, applying the first one that matched.[22] We will call the resulting compiler LLVM+Alive.

Experiments were performed on an Intel X68-64 machine running Ubuntu 14.04. Compilation was performed with optimization level 3. On average, LLVM+Alive compiled the SPEC 2000 and SPEC 2006 benchmarks 7% faster than LLVM. This is most likely because LLVM+Alive excludes two-thirds of the optimizations in InstCombine, meaning that fewer transformations were applied and fewer secondary opportunities to transform were created. In particular, LLVM+Alive included no transformations targeting `getelementpointer` instructions, which is the opcode LLVM most frequently transformed when compiling the benchmarks.

The benchmark programs compiled by LLVM 3.6 and LLVM+Alive were then executed on their reference inputs. LLVM+Alive-compiled code showed an average 3% slowdown across the SPEC benchmarks. Performance on individual benchmarks varied, with LLVM+Alive-compiled code faster by 7% on the gcc benchmark and slower by

[22]This matches the structure of InstCombine, which only attempts to optimize an add instruction with transformations that apply to adds. Earlier versions of the generated code did not group the transformations in this way, resulting in slower performance.

10% on the equake benchmark. Some slowdown was expected, given that LLVM+Alive performed fewer optimizations than LLVM 3.6.

During testing, LLVM+Alive consistently entered an infinite loop for specific benchmarks. The cause of this was eventually determined to be optimizations whose preconditions were weakened during translation, because they were stronger than needed for correctness. The stronger preconditions were needed to prevent a transformation from repeatedly applying to its own output, or to prevent one transformation from undoing another. These transformations were corrected or removed and are not reflected in the performance numbers. Later conversation with LLVM developers revealed that non-termination in InstCombine was a problem that occasionally cropped up and was difficult to diagnose. This inspired the work on non-termination checking discussed in Chapter 5.

3.7.3 Adoption by developers

Alive is commonly used by LLVM developers to evaluate the correctness of proposed peephole optimizations and has been used to detect several incorrect transformations at the time of proposal. For example, in July 2014, a developer proposed an optimization that improved performance for a SPEC benchmark by 3.8%. While this transformation passed LLVM's regression tests, it was proven incorrect after translation to Alive. After some discussion, its precondition was strengthened sufficiently to make the transformation correct and the patch was accepted to LLVM [54].

Recent work to simplify the semantics of LLVM [65, 74] created a prototype version of LLVM that implemented the proposal and used a modified version of Alive to check its optimizer. Using a tool, they generated programs involving up to three instructions, optimized them using the prototype, and then used Alive to test whether the optimized code refined the original.

The goal of Alive was to create a system for automated verification of LLVM peephole optimizations that would be usable by mainstream developers. Alive now has several open source implementations [75, 84] and an on-line service provided on the web [1]. Alive is now sufficiently well-known that a brief outage of its web interface

lead to developers asking on the LLVM mailing list asking for its status [121]. It would be fair to say Alive has met its goal.

3.8 Summary

Alive is a language for specifying peephole transformations for LLVM IR that has been designed to enable automated correctness checking of transformations and translation into C++ code suitable for use in LLVM. In particular, Alive can reason about undefined behavior present in the code being transformed and in the transformation process itself. Several hundred transformations included in LLVM have been translated into Alive and checked for correctness, uncovering several bugs in the process. More importantly, Alive is now used by LLVM developers to check the correctness of proposed new transformations for the peephole optimizer. By carefully designing a language, the Alive project has created a verification tool that has the power of formal methods while remaining usable by mainstream developers.

Chapter 4

Automated Precondition Inference

Alive provides a way to specify transformations of LLVM IR, and the Alive toolkit will automatically determine whether these transformations are correct. To aid developers who have written an incorrect transformation, the toolkit will report a counter-example, showing a case where the transformation would change the behavior of a program, either by computing a different result or by introducing undefined behavior. The developer can then create or strengthen the precondition for that transformation to exclude the cases where it is incorrect.

For example, given the transformation in Figure 3.21(a), Alive will report that the transformation is incorrect if `%X` is poison and `C` is 1, because the source will return poison but the target will have undefined behavior. This is because the target will divide `%X` by -1, which has undefined behavior if `%x` is the minimum signed integer (see Table 3.9). This case can be avoided by adding a precondition `C != 1`, but it is not sufficient to make the transformation correct. If `%X` and `C` are both the minimum integer, the source will return -1 but the target will return 1. This is because negating the minimum integer overflows, returning the minimum integer. To avoid this case, the precondition can be strengthened to to `C != 1 && !isSignBit(C)`.[1] With this precondition, Alive will report that the transformation is correct.

Unfortunately, debugging an incorrect transformation can be considerably more challenging. Figure 4.1(a) shows an incorrect transformation that was proposed to the LLVM developers mailing list for addition into LLVM. It passed LLVM's regression tests and showed a small but significant speed-up on one of the SPEC 2006 benchmarks. Unfortunately, Alive showed it to be incorrect, even when the precondition was

[1]The `isSignBit(C)` predicate from LLVM is true if and only if `C` is the minimum signed integer.

```
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
=>
  %a = and %A, ~(C1 ^ C2)
  %z = add %a, umax(C1, C2)
  %r = icmp ult %z, C3
```

(a) An incorrect transformation proposed for LLVM

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

(b) A precondition that makes the transformation correct

Figure 4.1: Finding a good precondition for a transformation can be difficult. It took several attempts to find (b), a precondition that makes (a) correct [54]. However, this precondition is both unusually complex and admits only a fraction of the cases where the transformation is correct.

strengthened to require `C1 u> C3` and `C2 u> C3`. Eventually, with the assistance of Alive and an expert on LLVM semantics, the developer found the precondition shown in Figure 4.1(b) [54], which is sufficient to make the transformation correct.

Finding a precondition that makes a transformation correct is important, but it is not the only factor that must be considered. In some cases, a precondition will make a transformation correct but prevent it from applying in some cases where it would be safe. For example, when using 1-bit arithmetic, the transformation in Figure 3.21(a) is correct when `C` is 1, but this is rejected by the suggested precondition. Similarly, the precondition in Figure 4.1(b) prevents the transformation from being applied in many circumstances where it would be safe to do so.

The complexity of a precondition must also be considered. If the precondition for Figure 3.21(a) is weakened to permit the 1-bit case where `C` is 1, it becomes at least 50% larger: `(C != 1 && !isSignBit(C)) || width(C) == 1`. Large preconditions with many nested clauses are difficult to understand and may slow compilation.

A developer creating a transformation must find an precondition that makes the transformation correct, but occasionally must choose between a widely applicable precondition and a concise, understandable precondition. This chapter describes Alive-Infer, a method for automatically generating preconditions that make a transformation

correct. The only input required is the transformation itself. Alive-Infer generates many preconditions, some brief, others attempting to accept as many programs as possible. Each precondition is combined with the transformation and verified: only preconditions that make the transformation correct are reported to the user. Alive-Infer halts once it finds a precondition that makes the transformation correct and is proven to allow the transformation to apply in every situation where it would be correct.

Alive-Infer can be used to debug incorrect transformations, by finding a sufficiently-strong precondition, or it can be used to generalize correct transformations, by finding preconditions that accept more programs while keeping the transformation correct.

4.1 Predicates and Preconditions

A *precondition* is part of a transformation that restricts the cases where it can apply. The precondition is a *predicate*, or Boolean-valued expression using variables defined by the source of the transformation. Predicates follow the grammar described by the *pre* production in Figure 3.3. They can be divided into two groups: *compound predicates*, which are constructed from logical connectives and other predicates, and *atomic predicates*, which include **true**, comparisons, and predicate functions.

Predicates, and preconditions, can be partially ordered according to strength. Two preconditions have the same strength if they are equivalent. That is, predicates P and Q have the same strength if

$$\forall_{\mathcal{V}} : P \iff Q. \tag{4.1}$$

For example, the predicates `C1 u<= 1` and `C1 >= 0 && C1 <= 1` are equivalent.

A predicate P is stronger than a predicate Q if P rejects whenever Q rejects but does not accept whenever Q accepts. If P is stronger or equivalent to Q, it will accept whatever Q accepts, that is

$$\forall_{\mathcal{V}} : P \implies Q. \tag{4.2}$$

For example, `C1 != 1` is stronger than `C1 > 1`.

It is possible that P is not equivalent, weaker than, or stronger than Q, in which case they are unordered. For example, `C1 < C2` and `C1 u< C2` are not ordered.

For each transformation T there is a set of predicates that could be its precondition, which is determined by the grammar of Figure 3.3 as well as the variable scoping and type correctness rules. Depending on the predicate used for its precondition, T may be correct or incorrect. If T is correct when its precondition is a predicate P, we say that P is a *sufficient precondition*. Not all sufficient preconditions are equally useful. The strongest predicate, `!true`, is always sufficient, because it makes T trivially correct by preventing it from ever being applied. To be useful, a precondition should be weaker than `!true`.

A sufficient precondition P is a *full precondition* if no weaker sufficient preconditions exist. A full precondition allows T to apply to every program where applying T is a refinement. It is sometimes referred to as the weakest precondition, but it is not guaranteed to be unique.

Any sufficient precondition that is not full is a *partial precondition*. Such a precondition will prevent T from applying to some programs where it would be a refinement. Occasionally, a partial precondition will be more concise than a full precondition while still accepting sufficiently many programs to be useful.

4.2 Precondition Inference

The goal of Alive-Infer is to generate sufficient preconditions for a transformation, including concise partial preconditions and a full precondition. This could be accomplished in several ways. A simple strategy is to enumerate candidate preconditions in non-decreasing order of size, checking each one to see whether it is sufficient. Assuming that a finite sufficient precondition exists, this method is guaranteed to eventually find it. Unfortunately, this process may require an enormous amount of time: the transformation will need to be verified for each enumerated precondition, and the number of preconditions grows exponentially with size.

Frequently, complex preconditions are compound predicates containing several different atomic predicates linked with logical connectives. Because the search space grows

exponentially with size, enumerating several small predicates is much faster than enumerating one large, compound predicate—even if enumeration starts from the beginning for each predicate. The Predicate Inference Engine (PIE) [96] takes advantage of this by learning preconditions in two phases. First, it learns predicates through enumeration, then it finds a formula using these predicates that will become the precondition.

In order to learn predicates, PIE must be able to assess their likely usefulness. It does this by generating positive and negative examples, representing concrete situations that the desired precondition must accept or reject, respectively. When learning predicates, it looks for ones that help separate the positive and negative examples. Once enough predicates have been learned, it finds a minimal formula using them that accepts all positive examples and rejects all negative examples.

Alive-Infer is inspired by the approach of PIE, but adapts it to Alive and makes several additions. First, unlike the programs PIE reasons about, Alive transformations have two kinds of variables: symbolic constants, which have values known to the compiler, and input variables, which are not known to the compiler. The precondition must accept or reject without knowing the values of the input variables, so determining whether an example is positive or negative will involve universal quantification. Second, Alive transformations, including preconditions, are type parametric. Care must be taken to ensure that predicates are type correct for all assignments of types to the precondition. Moreover, Alive-Infer must be able to infer preconditions such as `width(%x) u> 1`, which vary based on type assignments alone. Third, PIE only reasons about predicates that are defined for all argument values, but Alive frequently uses predicates such as `C1 % C2 == 0`, which are not defined for some examples. Alive-Infer reasons about partially-defined predicates and ensures that all generated preconditions are safe (*i. e.*, do not have undefined compile-time behavior). Finally, Alive-Infer generates both full preconditions and partial preconditions. These use two different formula learners, which are oriented towards accepting all positive examples and maintaining a bound on formula complexity, respectively.

Algorithm 4.1 shows the high-level structure of Alive-Infer. The inputs are a transformation T and an optional set of initial predicates P_0. The initial step is to derive

Algorithm 4.1 Infer a precondition for transformation T starting with predicates P_0

```
1:  procedure InferPrecondition(T, P_0)
2:      E^+, E^- ← GenerateExamples(T)
3:      P ← P_0                                          ▷ learned predicates P
4:      loop
5:          φ, P ← InferByExamples(T, P, E^+, E^-)       ▷ candidate precondition φ
6:          e^- ← FindCounterExamples(T, φ)
7:          if e^- ≠ ∅ then
8:              E^- ← E^- ∪ e^-
9:          else
10:             φ ← EnsureSafety(T, E^+, E^-, φ)
11:             e^+ ← FindPositiveExamples(T, φ)
12:             if e^+ = ∅ then
13:                 Report full precondition φ
14:                 return
15:             else
16:                 Report partial precondition φ
17:                 E^+ ← E^+ ∪ e^+
18:             end if
19:         end if
20:     end loop
21: end procedure
```

positive and negative examples from T, using the example generation methods described in Section 4.3. Next, it repeatedly infers candidate preconditions ϕ until it finds one that is sufficient and necessary to make T correct. Line 5 finds ϕ and updates the set P of predicates with any new predicates learned by InferByExamples. The candidate will reject all examples in E^-, but may accept unknown negative examples. Line 6 searches for these examples, adding them to E^- if they exist. If none exist, ϕ is sufficient to make T correct and will be reported to the user as a full or partial precondition, depending on whether ϕ rejects any known or unknown positive examples. Line 11 searches for positive examples rejected by ϕ. If none exist, then ϕ is reported as a full precondition and InferPrecondition halts. Otherwise, ϕ is reported as a partial precondition. Any new positive examples are added to E^+ and inference continues.

Note that InferPrecondition only reports preconditions that are sufficient to make T correct. The procedure halts once it finds a precondition that is both sufficient and necessary, meaning it accepts all possible positive examples (with some possible exceptions, as discussed in Section 4.3.2).

The prototype implementation in Alive-NJ follows this basic outline, but has additional details. For example, InferPrecondition only checks for additional positive examples when the generated precondition is a candidate full precondition. Structurally, InferPrecondition and InferByExamples are implemented as co-routines, permitting InferByExamples to retain internal data structures and avoid generating any precondition more than once.

The subroutine FindCounterExamples simply verifies the correctness of T with precondition ϕ, using the methods described in Chapter 3, but augmented to return multiple counter-examples for incorrect transformations. Unlike normal verification, FindCounterExamples assumes that ϕ is always safe by additionally asserting its safety condition σ_p. Once the lack of counter-examples has been confirmed, the safety assumption is made explicit by EnsureSafety, which is described in Section 4.5.3.

Finally, FindPositiveExamples finds positive examples for T that are rejected by ϕ. This is an extension of the method for finding positive examples discussed in Section 4.3.

4.3 Generating Examples

The core of Alive-Infer is data-driven precondition inference. The algorithm uses two sets containing positive and negative examples, respectively, and learns a precondition that accepts the positive examples and rejects the negative examples. In PIE, an example is an assignment of values to variables, but Alive has two kinds of variables, only one of which is available to the compiler. Rather than representing a program state which may or may not satisfy a precondition, examples in Alive-Infer represent abstract program fragments where applying a transformation may or may not always refine its behavior.

The precondition is only evaluated when the input code matches the transformation's source pattern, so that structure can be assumed. What remains are the parts of LLVM IR that Alive abstracts, such as types and symbolic constants. An example, then, is an assignment of concrete types to values and concrete values to symbolic

constants.[2] If the source were to be made concrete using these assignments, then the source could be used directly as LLVM IR (assuming it is embedded into an LLVM function, as in Section 5.4).

In addition to types and constants, Alive preconditions can also refer to the results of dataflow analyses. In principle, these could also be represented as part of the example, but each possible analysis would require separate results. If the source has three input variables, there will be three to six possible analyses for each of the dataflow predicate functions just using the variables. For the binary functions, the number of possible analyses grows quadratically with the number of input variables and symbolic constants. For this reason, Alive-Infer does not infer preconditions that use dataflow analyses. However, extending examples to include a subset of analysis results, possibly selected by the user, should be relatively simple and would require no changes to the core algorithm.

Note that dataflow predicates typically provide exact results if their arguments are symbolic constants. Because their behavior can be determined at compile time, they do not require additional variables in the example to represent their results, and can be enumerated in the same manner as non-dataflow functions.

4.3.1 Classification of examples

Each example represents a concrete code fragment. For *positive* examples, applying the transformation to that fragment will refine the behavior of the fragment. For *negative* examples, applying the transformation will change its result or introduce undefined behavior.

More concretely, an example for a transformation T is a pair $\langle \tau, \hat{c} \rangle$, containing a type assignment τ for the values in T and a value assignment $\hat{c}$ for $\mathcal{C}$, the symbolic constants in T. If, and only if, the correctness conditions given in Section 3.2.7 are satisfied for the encoding of T with types τ and substituting the values in $\hat{c}$ for the variables in $\mathcal{C}$, then $\langle \tau, \hat{c} \rangle$ is a positive example.

[2] As an optimization, the Alive-NJ prototype associates each value with a type variable. Examples then provide concrete types for type variables, rather than individual terms.

Because T has no precondition, the correctness conditions can be reduced to

$$\Psi \equiv \forall_{\mathcal{I},\mathcal{P},\mathcal{Q}_t} \exists_{\mathcal{Q}_s} : \chi_t \implies \sigma_t \wedge (\delta_s \implies \delta_t) \wedge (\delta_s \wedge \rho_s \implies \rho_t \wedge \iota_s = \iota_t). \tag{4.3}$$

An example $\langle \tau, \hat{c} \rangle$ is positive if and only if Ψ is satisfied using an encoding with types τ and substituting the variables in $\mathcal{C}$ according to $\hat{c}$.

4.3.2 Explicit assumptions

Not all examples are equally desirable. Recall the incorrect transformation in Figure 3.21(a), which is incorrect when `C` equals 1. To be correct, the precondition must forbid `C` to be 1, but, in the context of a larger transformation pass, this check is unnecessary. A higher-priority transformation will replace **sdiv** $x, 1$ with x before the corrected transformation can apply. Thus, only the requirement that `C` not be the minimum signed integer is needed to ensure that the peephole optimization pass as a whole is correct.

Often, transformations will have edge cases that would make the precondition more complex to handle appropriately, but which are guaranteed not to occur. Recall that the full precondition for Figure 3.21(a) needed to exclude `C` equal to 1 except when `C` is a 1-bit value, in order to allow an obscure case that, in the context of the whole optimizer, will never occur.

To avoid this unnecessary complexity, Alive-Infer allows transformations to state explicit *assumptions* about the examples they will encounter. Examples that violate these assumptions are neither positive or negative: the precondition is free to accept or reject them without affecting the correctness of the transformation or the generality of the precondition. In particular, a full precondition may reject positive examples do not satisfy the assumption. This means that two full transformations may not be equivalent if they differ for examples that violate the assumptions.

Like the precondition, an explicit assumption will be a predicate A using variables from the transformation. Its safety condition, σ_A, should be satisfied for all values of symbolic constants, and it should not depend on any dataflow analysis. We will write α for the encoding of A under some type assignment.

```
Assume: C != 1
Pre: !isSignBit(C)
  %a = sdiv %X, C
  %r = sub 0, %a
=>
  %r = sdiv %X, -C
```

Figure 4.2: The transformation from Figure 3.21(a), with an explicit assumption and a precondition that is full with respect to the assumption

In the concrete syntax, explicit assumptions can be provided using an `Assume:` header, which is syntactically similar to the precondition header. If no explicit assumption is provided, it defaults to **true**. Figure 4.2 shows the transformation from Figure 3.21(a) with an explicit assumption. Because the assumption excludes all cases where `C` is 1 from consideration, the precondition is full.

In addition to any explicit assumptions, Alive-Infer also assumes that the source is well-defined and poison-free for at least one assignment of the input variables. The correctness condition Ψ is trivially satisfied if the source is never well-defined and poison-free, but requiring the precondition to accept these examples can complicate it by introducing edge cases. Such an ill-defined program is likely to be removed by other optimization passes, so extra work spent to allow optimization of those programs is wasted.

Thus, all examples generated by Alive-Infer will satisfy this condition:

$$\Phi \equiv \alpha \wedge (\exists_{\mathcal{I},\mathcal{Q}_s} : \delta_s \wedge \delta_t). \tag{4.4}$$

In particular, positive examples will satisfy $\Phi \wedge \Psi$ while negative examples will satisfy $\Phi \wedge \neg\Psi$.

4.3.3 Generation methods

The goal of example generation is to find an initial set of examples that satisfy the assumptions Φ and to classify them as positive or negative, depending on whether they satisfy the correctness condition Ψ. Starting inference with a small number of examples, or none, is likely to result in InferByExamples generating an insufficient or partial precondition. The verification checks in InferPrecondition will detect this, and add

```
Pre: width(%r) != 1
  %r = add %x, %x
=>
  %r = shl %x, 1
```

Figure 4.3: Alive-Infer is able to infer the precondition shown for this transformation, even though it contains no symbolic constants

more negative or positive examples as needed, but finding examples using solver queries can be slow. A transformation with a complex boundary between positive and negative examples may require many examples to correctly guide the inference process. Starting inference with a large number of examples, generated using faster methods, will result in faster generation of better preconditions.

Alive-Infer includes two methods for generating the initial examples. Both methods assume a given type assignment τ. A transformation T may have many valid assignments, and including examples with different assignments will increase the likelihood of learning predicates that are type-dependent (*e. g.*, `width`). The Alive-NJ prototype enumerates type assignments (up to a limit on integer width) and chooses a subset that is roughly logarithmic in the total number of feasible assignments. It does this by skipping some assignments, and doubling the number of assignments to skip each time. The number of type assignments for a transformation is exponential in the number of independent type variables, so this logarithmic sampling provides a set of assignments roughly linear in the number of type variables.

Happily, the methods described below work even in the trivial case where $\mathcal{C}$ is empty. Each type assignment will still give rise to a separate example containing an empty value assignment $\hat{c}$, and the example generation methods will correctly determine whether this example is positive or negative. Figure 4.3 shows a transformation with a non-trivial precondition that involves only type information. It is incorrect in the 1-bit case, because any shifting of a 1-bit integer will overflow. Alive-Infer will generate multiple examples that differ only by type assignment, and then learn an appropriate precondition equivalent to the one shown in the figure.

The methods use SMT queries to classify or generate examples. These queries are based on the assumption filter Φ from Equation (4.4) and the correctness condition Ψ

from Equation (4.3), and produce positive examples that satisfy $\Phi \wedge \Psi$ and negative examples that satisfy $\Phi \wedge \neg\Psi$. Unfortunately, the quantifier alternation in Ψ becomes a double quantifier alternation when using $\Phi \wedge \Psi$ as a query for positive examples.

Random selection

There are several methods for generating value assignments. The simplest method is to randomly select values for each symbolic constant according to its assigned type, and then classify the resulting example as positive or negative using an SMT solver. For a given type assignment τ, $\hat{c}$ is created by randomly selecting values for each variable according to their type assigned by τ.

Each generated example $\langle \tau, \hat{c} \rangle$ is checked twice: first to see whether it satisfies the assumptions, and second to classify it as positive or negative. An example that does not satisfy the assumptions is discarded.

Writing α for the encoding of the explicit assumption, the assumption test checks whether α is satisfied by $\hat{c}$ and whether the source is well-defined and poison-free for at least one assignment of the run-time variables $\mathcal{I}$. The SMT solver is given the query

$$\exists_{\mathcal{I},\mathcal{Q}_s} : \alpha \wedge \delta_s \wedge \rho_s, \tag{4.5}$$

after the values in $\mathcal{C}$ have been replaced with concrete values according to $\hat{c}$. If the formula cannot be satisfied, the assumptions are violated and the example is discarded.

Examples that satisfy the assumptions must now be classified as positive or negative. An example is negative if some assignment to $\mathcal{I}$ exists that violates the correctness conditions from Section 3.2.7. The SMT solver is given the query

$$\exists_{\mathcal{I},\mathcal{P},\mathcal{Q}_t} \forall_{\mathcal{Q}_s} : \chi_t \wedge \neg\sigma_t \vee (\delta_s \wedge \neg\delta_t) \vee (\delta_s \wedge \rho_s \wedge (\neg\rho_t \vee \iota_s \neq \iota_t)), \tag{4.6}$$

after the values in $\mathcal{C}$ have been replaced with concrete values according to $\hat{c}$. If the formula is satisfiable, then $\langle \tau, \hat{c} \rangle$ is negative. Otherwise, it is positive.

Generation using a solver.

Random selection is relatively fast, but there is a risk that the examples it finds will be all positive or all negative. To ensure that at least some positive and negative examples

are found (or to determine that none exist, in which case the precondition is trivial), Alive-Infer uses a solver to generate positive and negative examples that satisfy the assumptions.

A negative example will satisfy the assumptions but violate the correctness conditions for some valuation of $\mathcal{I}$. The query for the SMT solver simplifies to

$$\exists_{\mathcal{C},\mathcal{I},\mathcal{P},\mathcal{Q}_t} \forall_{\mathcal{Q}_s} : \alpha \wedge \delta_s \wedge \rho_s \wedge \chi_t \wedge (\neg\sigma_t \vee \neg\delta_t \vee \neg\rho_t \vee \iota_s \neq \iota_t). \tag{4.7}$$

The model returned by the solver will include values for $\mathcal{C}$, which are used to construct $\hat{c}$.

Finding positive examples requires a more complex query, because the correctness condition must hold for all valuations of $\mathcal{I}$. Additionally, the source should be well-defined and poison-free for at least one valuation of $\mathcal{I}$. The resulting query is

$$\begin{aligned} &\exists_{\mathcal{C}} : \alpha \wedge (\exists_{\mathcal{I},\mathcal{Q}_s} : \delta_s \wedge \rho_s) \\ &\qquad \wedge [\forall_{\mathcal{I},\mathcal{P},\mathcal{Q}_t} \exists_{\mathcal{Q}_s} : \chi_t \implies \sigma_t \wedge (\delta_s \implies \delta_t) \wedge (\delta_s \wedge \rho_s \implies \rho_t \wedge \iota_s = \iota_t)]. \end{aligned} \tag{4.8}$$

Again, the model returned by the solver will provide values for $\mathcal{C}$, which are used to construct $\hat{c}$. The alternation of quantifiers may slow down the solver considerably, so this method is best used when $\mathcal{Q}_s$ is empty.

4.4 Predicate Learning

The data-driven inference method used by Alive-Infer has two parts: predicate learning and Boolean formula learning (discussed in Section 4.5). The former finds a set of atomic predicates that may be useful to separate the positive and negative examples, and the latter uses the predicates to assemble a precondition. These stages may be performed sequentially, as in PIE, first learning enough predicates to completely separate the positive and negative examples, and then learning the formula using the predicates. Instead, Alive-Infer interleaves these stages, generating a partial precondition each time a new predicate is learned. This allows Alive-Infer to begin reporting suggested preconditions to the user before the learning process has completed.

Algorithm 4.2 outlines this process. The procedure InferByExamples is given a transformation T, a set of previously learned predicates P, and two sets of examples

E^+ and E^-. It first searches for a *mixed group* of positive and negative examples, such that no predicate in P will have different results for any two examples in the group. If no mixed group exists, then it uses P to create a full precondition. Otherwise, it uses the group to learn a new predicate by enumerating predicates until it finds one that separates a subset of the group no larger than a particular threshold. This new predicate is added to P and the procedure generates a partial precondition (which may be the trivial partial precondition `!true`). The procedure returns the candidate precondition ϕ and the augmented set of preconditions P'.

Algorithm 4.2 Infer a full or partial precondition for T using examples E^+, E^- and predicates P. Return the precondition and a new set of predicates.

```
1:  procedure InferByExamples(T, P, E+, E−)
2:      e+, e− ← ChooseMixedGroup(P, E+, E−)
3:      if e+ = ∅ then                          ▷ The predicates in P are sufficient
4:          φ ← LearnFullFormula(P, E+, E−)
5:          return φ, P
6:      end if

7:      e+_s, e−_s ← SampleGroup(e+, e−)
8:      for p ∈ Predicates(T) do
9:          if Separates(p, e+_s, e−_s) ∧ Safe(p, E+) then        ▷ Learn predicate p
10:             P′ ← P ∪ {p}
11:             φ ← LearnPartialFormula(P′, E+, E−)
12:             return φ, P′
13:         end if
14:     end for
15: end procedure
```

The prototype implements a variation of this algorithm with a few additional details. The procedures InferPrecondition and InferByExamples are implemented as co-routines that pass each other examples and candidate preconditions, respectively. This adds a loop to InferByExamples and allows it to avoid generating trivial preconditions or passing any candidate precondition to InferPrecondition more than once. Additionally, it maintains a data structure for efficiently finding mixed groups. These differences are optimizations and do not increase the power of the procedure.

4.4.1 Predicate behavior

Predicate learning in Alive-Infer focuses on atomic predicates, such as comparisons and predicate functions. Complex predicates involving logical connectives (and, or, not) will be generated later by the formula learner (see Section 4.5).

Predicate learning operates by observing the *behavior* of predicates for particular examples. This corresponds to the execution of the corresponding C++ implementation when transforming the corresponding IR fragment. For a particular example, a predicate may be true or false, but recall that constant expressions (and thus predicates involving constant expressions) may be undefined for certain values. If a predicate's safety condition is not satisfied for an example, we say the predicate is *unsafe* for that example.

More concretely, we will write $B(p, e)$ for the behavior of predicate p for an example $e = \langle \tau, \hat{c} \rangle$ and the possible behaviors as $\top$ (true), $\bot$ (false), and $\star$ (unsafe). To find $B(p, e)$ we take the safety condition (σ_p) and value (ι_p) from the SMT encoding of p for types τ and substitute the values in $\hat{c}$ for the symbolic constants. If σ_p is not satisfied, then $B(p, e) = \star$. Otherwise, $B(p, e) = \iota_p$.

While it is acceptable for a predicate to be unsafe for a particular example, the precondition must always be safe. (Recall the first correctness condition from Section 3.2.7.) One way to avoid unsafe preconditions is to forbid unsafe predicates, but this prevents Alive-Infer from using useful predicates such as `C1 % C2 == 0`. Another way is to make the formula learner aware of the non-commutative semantics for Alive's conjunction and disjunction operators, but this complicates the work the formula learner must perform and does not guarantee that the resulting formula is safe.

Rather than modify the formula learner, it is also possible to modify the preconditions it learns to avoid unsafe behavior. The formula learners produce formulae in the form $p_1 \wedge p_2 \wedge \cdots p_n$, where the p_i may be atomic predicates or disjunctions. Any unsafe negative examples can be pre-rejected by adding a prefix clause s_0 to the beginning. If a clause p_i is unsafe for any positive examples, those examples can be pre-accepted by a sub-clause s_i, resulting in a formula of the form $s_0 \wedge (s_1 \vee p_1) \wedge \cdots \wedge (s_n \vee p_n)$.

The various s_i can be found by repeating the inference process. While inference of the safety prefixes is likely to be much faster than precondition inference, this process is, cumbersome and may produce larger or even non-CNF formulae.

Alive-Infer compromises by restricting unsafe behavior to negative examples. If $B(p, e) = \star$ for any positive example e, we reject p as unsuitable. This simplifies the design of the predicate and formula learners, because it is always the case that unsafe behavior for an example means that example should be rejected. Once a precondition has been found, any negative examples that cause unsafe behavior can be filtered out by adding a single safety prefix (see Section 4.5.3).

4.4.2 Grouping examples by behavior

Given a set of predicates P and two examples e_1 and e_2, we say that P separates e_1 and e_2 if there is a predicate $p \in P$ such that $B(e_1, p) \neq B(e_2, p)$. Otherwise, we say e_1 and e_2 have the same behaviors for P. Given sets of positive and negative examples E^+ and E^-, we can use P to group examples that have the same behaviors. A group may contain only positive examples, only negative examples, or a mixture of positive and negative examples. We refer to the latter as *mixed groups*. In Figure 4.4, the first three examples form a mixed group.

If the predicates in P are ordered, then we may define a *behavior vector*, which describes the behaviors of an example. The behavior vector V for an example e and predicates P is a list of behaviors, such that $V_i = B(e, p_i)$. The examples in a particular group will have equal behavior vectors, and each group will be associated with a different vector. The rightmost columns of each row in Figure 4.4(c) give behavior vectors such as $\top\top\top$ for the first three examples.

Efficiently finding mixed groups

A key subroutine in Algorithm 4.2 is ChooseMixedGroup, which returns the positive and negative examples in an arbitrary mixed group, or two empty sets if no mixed group exists. Rather than repeatedly searching for a mixed group, the prototype maintains a data structure that groups examples with the same behavior for all predicates learned

```
%m = mul nsw %X, C1
%r = sdiv %m, C2
=>
%r = sdiv %X, C2 / C1
```

(a) A transformation

p_1 `C2 % C1 == 0`
p_2 `C2 == -C1`
p_3 `C2 < 0`

(b) Learned predicates

Example				Behavior		
τ	c_1	c_2	$\pm$	p_1	p_2	p_3
i4	2	-2	$-$	$\top$	$\top$	$\top$
i4	1	-1	$+$	$\top$	$\top$	$\top$
i4	-8	-8	$+$	$\top$	$\top$	$\top$
i4	-1	1	$-$	$\top$	$\top$	$\bot$
i4	2	-2	$+$	$\top$	$\bot$	$\top$
i4	2	4	$+$	$\top$	$\bot$	$\bot$
i4	2	-1	$-$	$\bot$	$\bot$	$\top$
i4	2	3	$-$	$\bot$	$\bot$	$\bot$
i4	0	1	$-$	$\star$	$\bot$	$\bot$

(c) Behavior of predicates

Figure 4.4: An intermediate step during inference. InferByExamples has learned three predicates for the transformation in (a), which are shown in (b). A selection of examples are given in (c) along with their classification as positive or negative and their behavior for the learned predicates.

up to that point. The structure represents each group as a triple containing the sets of positive and negative examples and the behavior vector common to all examples in the group. Finding a mixed group is simply a matter of finding a triple with two non-empty sets.

Each time a new predicate is learned, the groups are subdivided according to their behavior for the new predicate. The behaviors for the old predicates do not need to be re-evaluated, as they are given by behavior vectors.

On occasion, InferPrecondition will add new positive or negative examples. The behaviors of these examples for the learned predicates are determined, and they are added to the groups. However, if any of the learned predicates are unsafe for a new positive example, that predicate must be removed in order to maintain the invariant that no predicate is unsafe for any positive example. (This is not shown in Algorithm 4.2.)

4.4.3 Learning new predicates

If the current set of predicates P is sufficient to divide the positive and negative examples in E^+ and E^-, meaning there are no mixed groups, then P is sufficient to construct a full precondition. Otherwise, Alive-Infer will learn a new predicate with the intention of separating the positive and negative examples in the mixed groups.

As shown in Algorithm 4.2, Alive-Infer selects a single mixed group with the procedure ChooseMixedGroup, chooses a subset of that group no larger than a threshold with SampleGroup, and then enumerates predicates until it finds one that separates the positive and negative examples in the sample (Separates). This predicate is added to P, and Alive-Infer uses the expanded P to learn a partial precondition that rejects all negative examples. (If no positive-only groups exist, the best partial precondition is `!true`. Rather than generate these trivial preconditions, the Alive-Infer prototype waits until at least one positive example is separated from all negative examples before learning preconditions.)

Selecting a mixed group

By using examples from a single mixed group, Alive-Infer has a stronger guarantee of termination. If, instead, Alive-Infer selected examples from multiple mixed groups, it is possible that Alive-Infer would repeatedly learn the same predicate. For example, learning a predicate p might subdivide a mixed group e^+, e^- into e^+_p, e^-_p (which satisfy p) and $e^+_{\neg p}, e^-_{\neg p}$ (which do not satisfy p). When learning the next predicate, Alive-Infer will choose one group or the other, guaranteeing that whatever predicate is found will not be p because every example in the two groups has identical behavior for p.

Finding mixed groups is simple, using the data structure discussed in Section 4.4.2, but some heuristic is needed when choosing one of multiple mixed groups. By default, the prototype chooses the largest mixed group, but other strategies are possible, such as selecting the smallest group, or the group containing the most positive examples. It is not clear how the choice of strategies affects the average time needed to learn predicates.

Sampling the mixed group

Mixed groups may be very large. In particular, when P is initially empty, the first mixed group contains *all* initial examples. To increase the chances that a simple predicate will separate the examples, SampleGroups ensures that $|e^+| + |e^-|$ does not exceed some threshold by randomly selecting examples from the mixed group, with the requirement

that the sample contain at least one positive and negative example. If the mixed group is smaller than the threshold, then the entire group is used. The prototype uses a threshold of 16, but experiments with other thresholds are recommended.

A possibility remains that the examples in the sample require a large predicate to separate them, greatly increasing the time needed to learn a predicate. To reduce this risk, the prototype draws multiple samples and observes each enumerated predicate for each of the samples until it finds a predicate that separates at least one.

Finding a predicate

Given a sample of positive and negative examples e^+, e^-, Alive-Infer enumerates predicates until it finds one that (a) separates e^+ from e^- and (b) is safe for all examples in E^+. Enumeration is restarted each time a new predicate is learned, because predicates that were not able to separate a sample from a larger group may be able to separate samples from a smaller group. In Figure 4.4, learning `C2 % C1 == 0` created smaller groups where it was more likely that `C2 == -C1` would separate a sample.

A predicate p separates the sample e^+, e^- if $B(p, e_1) \neq B(p, e_2)$ for all examples $e_1 \in e^+$ and $e_2 \in e^-$. That is, p is true for all examples in e^+ and false or unsafe for all examples in e^-, or p is false for all examples in e^+ and true or unsafe for all examples in e^-. (Because $e^+ \subseteq E^+$, we cannot use p if $B(p, e) = \star$ for any $e \in e^+$.)

Note that p is permitted to be false for the positive examples. This halves the number of predicates that must be enumerated, as p effectively stands for itself and for $\neg p$. The formula learner will ultimately determine whether to use either or both of p and $\neg p$ in the precondition.

4.4.4 Predicate enumeration

The predicate enumerator used by Alive-Infer generates type-correct predicates in non-decreasing order of size. The generated predicates are specific to a transformation, involving the symbolic constants and type variables used by a transformation and satisfying the type constraints for that transformation. (In Algorithm 4.2, this is represented by passing the transformation T to Predicates. The prototype instead derives

a configuration for the enumerator once and then re-uses it each time the predicates are enumerated.)

While Algorithm 4.2 shows PREDICATES returning an infinite set of predicates, the prototype structures the enumerator as a co-routine that generates predicates on-demand. Other designs are possible, but the co-routine structure neatly separates the enumerator from the predicate learner.

Each predicate has a *size*, which corresponds roughly to the number of AST nodes in its representation. Enumeration begins with size one, and all predicates of size one are enumerated before continuing to size two, and so forth. There is no upper limit on the size of a predicate, so enumeration can continue indefinitely.

Predicates are enumerated using a subset of Alive's predicate language, containing comparisons of constant expressions, selected predicate functions over constant expressions, and `width`. Constant expressions may be symbolic constants, expressions involving binary or unary operators, certain constant functions, and the literals `0` and `1`. Constant expression enumeration is parameterized by the size and type of the desired expression.

A naïve AST-based enumerator will generate many predicates that are ill-typed, have type ambiguities, or require stronger type constraints than T. To avoid wasting effort generating bad predicates or expressions, the enumerator has built-in knowledge of the typing constraints of Alive. Constant expressions are generated for a particular type, and indicate whether their type can be unambiguously derived from their content. For example, the enumerator will never generate `log2(1)`, because the type of `1` is ambiguous. Similarly, the enumerator will generate `C1 + zext(C2)` only if `C2` is already constrained to have a smaller bit width than `C1`.

The enumerator also uses knowledge of the language semantics to avoid generating redundant predicates. For example, several binary operators in Alive are commutative and associative, so the enumerator generates expressions in a normal form where it is practical to do so. The enumerator is also aware of distributivity, so it will generate $a \times b + a \times c$ but not $a \times (b + c)$. To avoid generating both $a < b$ and $b > a$, the enumerator does not allow the right side of the comparison to be a larger expression

than the left. If both sides are the same size, the right side must come from earlier in the enumeration than the left.

4.5 Formula Learning

Once Alive-Infer has learned a set of predicates, it uses a formula learner to construct a precondition using these predicates. The formula learners are designed to produce concise formulae, subject to certain restrictions. They automatically discard predicates that are not necessary to separate the positive and negative examples, and correctly determine whether to use a predicate or its negation.

Alive-Infer includes two formula learners. The full formula learner (discussed in Section 4.5.1) finds a precondition that accepts all positive examples and rejects all negative examples, but may be highly complex. The weighted partial formula learner (discussed in Section 4.5.2) finds a precondition that meets a complexity bound and rejects all negative examples, but may reject some positive examples.

Rather than working directly with examples, both learners receive sets of positive and negative behavior vectors (see Section 4.4.2). These represent groups of examples that are not separated by the set of learned predicates. Because no formula using these predicates can distinguish the examples in a group, the actual examples are not needed when finding a formula. A behavior vector is considered negative if its corresponding group contains any negative examples. Otherwise, it is positive.

Similarly, the formula learners do not need to know the specific predicates being used. The predicates can be treated as indices that select an element from a behavior vector, and formulae can be created using the familiar logical connectives extended to work with $\star$. For example, $\neg\star = \star$, and $\wedge$ and $\vee$ have a short-circuit semantics where the second argument is only evaluated if necessary, as seen in Figure 4.5. Once the formula learner has found a suitable formula, it is translated into Alive AST and the abstract predicates are replaced with the actual predicates.

When evaluating a formula with respect to a behavior vector, a result of $\top$ means accept, and $\bot$ and $\star$ mean reject. The learners find formulae that accept (some or all)

$r \wedge c$	$\top$	$\bot$	$\star$
$\top$	$\top$	$\bot$	$\star$
$\bot$	$\bot$	$\bot$	$\bot$
$\star$	$\star$	$\star$	$\star$

(a)

$r \vee c$	$\top$	$\bot$	$\star$
$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\star$
$\star$	$\star$	$\star$	$\star$

(b)

Figure 4.5: Short circuit operators $\wedge$ and $\vee$ in the presence of unsafe behavior ($\star$)

positive vectors and reject (all) negative vectors. This may lead to an Alive precondition that is unsafe for some examples. Section 4.5.3 describes a method for inferring additional clauses that reject all examples for which the predicate is unsafe.

4.5.1 Full formula learning

The full Boolean formula learner attempts to find the smallest formula that accepts all provided positive behavior vectors and rejects all negative vectors. The generated formulae are in conjunctive normal form (CNF), a conjunction of disjunctions of *literals*, which represent the predicates and their negations. The literal $\neg p_i$ indicates the negation of the ith predicate.

Algorithm 4.3 shows the structure of the learner. Its arguments are the number of predicates, n, and two disjoint sets of positive and negative vectors, V^+ and V^-. Each vector in V^+ and V^- has length n. The behavior of a predicate p_i on a vector v is v_i.

The learner has two parts. The first part finds a set of clauses C, where each clause in C accepts every positive vector. Each clause is a disjunction of up to k literals. The clause size starts at one and increases until every negative vector is rejected by at least one clause.

The second part finds a minimal subset of C that rejects all negative vectors. This is implemented as a greedy set-cover algorithm that selects clauses from C until every negative vector is covered (*i. e.*, rejected). Each selected clause is the member of C that increases the cover by the largest amount. If there is a tie, the prototype prefers smaller clauses.

By starting with $k = 1$ and increasing until all negative vectors are rejected, the first part ensures that the disjunctive clauses occurring in the formula are as small as

Algorithm 4.3 Full Boolean formula learner

```
procedure LearnFullFormula(n, V⁺, V⁻)
    L ← {p₁, ¬p₁, ..., pₙ, ¬pₙ}
    k ← 0
    C ← ∅
    while ∃_{v∈V⁻} : accepts(⋀C, v) do
        k ← k + 1
        C_k ← {⋁ d : d ⊆ L, |d| = k}                          ▷ disjunctions of size k
        C ← C ∪ {d : d ∈ C_k, ∀_{v∈V⁺} accepts(d, v)}   ▷ ... that accept all pos. vectors
    end while

    return MinimalCover(C, V⁻)
end procedure

procedure MinimalCover(C, V⁻)
    φ ← ⊤
    while ∃_{v∈V⁻} : accepts(φ, v) do
        c ← argmax_{d∈C} |{v : v ∈ V⁻, ¬accepts(d, v)}|
                                  ▷ pick clause that rejects the most remaining vectors
        V⁻ ← V⁻ \ {v : v ∈ V⁻, ¬accepts(d, v)}          ▷ remove rejected vectors
        C ← C \ {c}
        φ ← φ ∧ c
    end while

    return φ
end procedure
```

possible. The second part further reduces the formula size by minimizing the number of disjunctive clauses, although it is not guaranteed to find the minimum set.

Given n predicates, the number of clauses in C_k grows exponentially with k. The prototype saves some effort by not generating clauses containing a predicate and its negation, but $|C_k|$ remains exponential. If n is large, then the number of clauses that must be examined even at small values of k will also be large. This can occur if Alive-Infer has learned (or been provided) with many predicates that are not needed for the final precondition. This might be mitigated by filtering out predicates that do not contribute to separating the positive and negative examples before calling the formula learner.

Unfortunately, avoiding clauses containing a predicate and its negation can prevent the learner from finding a formula: because predicates can be unsafe for a vector,

we cannot assume $p \vee \neg p = \top$. Instead, such a clause is accepted only when the safety condition for p is satisfied. It is possible that the behavior vectors for a set of predicates implicitly rely on the safety conditions for one or more predicates. As a minimal example, there may be a single predicate and positive vectors $\top$ and $\bot$ and negative vector $\star$: this would imply the formula $p_1 \vee \neg p_1$, but the optimization in the previous paragraph prevents the learner from considering that clause.

It is possible to extend the learner to handle such cases by adding clauses of the form $p_i \vee \neg p_i$. Larger disjunctions involving a predicate and its negation will never be necessary, because the safety condition is guaranteed to be satisfied by all positive examples. The prototype does not currently implement this extension, instead signalling an error if it is unable to find a formula. Our experience has been that this situation occurs rarely, if ever, in practice.

4.5.2 Weighted partial formula learning

The weighted partial Boolean formula learner is a variation of the full formula learner that attempts to find a simple precondition that rejects all negative examples and accepts as many positive examples as possible.

Algorithm 4.4 shows the structure of the weighted, partial learner. Its arguments are the number of predicates, n, a set of weighted positive vectors, V_w^+, a set of negative vectors, V^-, and a maximum clause size K. The weighted vectors are pairs $\langle w, v \rangle$ comprising a behavior vector v and an integer weight w. Alive-Infer uses the size of the vector's group as the weight, but other weighting schemes are possible.

The algorithm begins by generating all disjunctive clauses up to size K. In the prototype, $K = 1$, but other choices are possible. It then selects clauses, adding them to the set C, until every negative vector is rejected by at least one clause in C. When selecting a clause, the algorithm chooses the clause that accepts the most weight (*i. e.*, rejects the fewest remaining positive examples). Once a clause is selected, any positive vectors it rejects are removed.

If the the algorithm is able to find a set C of clauses that cover the negative vectors, it finds a minimal cover using MinimalCover.

Algorithm 4.4 Weighted partial Boolean formula learner

```
procedure LearnPartialFormula(n, V_w^+, V^-, K)
    L ← {p_1, ¬p_1, ··· , p_n, ¬p_n}
    D ← {⋁ d : d ⊆ L, |d| ≤ K}                    ▷ all disjunctions up to size K
    C ← ∅                                          ▷ chosen clauses

    while ∃_{v∈V^-} : accepts(⋀ C, v) do
        c ← argmax_{d∈D} ∑{w : ⟨w, v⟩ ∈ V_w^+, accepts(d, v)}
                                    ▷ choose clause that accepts the most weight
        V_w^+ ← {⟨w, v⟩ : ⟨w, v⟩ ∈ V_w^+, accepts(⋀ C, v)}
                                                ▷ discard any vectors rejected by c
        C ← C ∪ {c}
        D ← D \ {c}
        if D = ∅ then
            return ⊥
        end if
    end while

    return MinimalCover(C, V^-)
end procedure
```

Algorithm 4.4 is sensitive to the initial choice of clauses, as the removal of vectors will reduce the weight accepted by the remaining clauses. The prototype extends the algorithm in an attempt to find alternative formulae that may accept more weight. After a formula is found, the prototype returns to the initial set of weighted positive vectors and begins choosing clauses again, but making sure to start with a clause that was not previously chosen. On occasion, this will produce a formula that accepts more weight.

Other designs for the partial learner are possible. Algorithm 4.4 chooses clauses based on the weight of the positive vectors it accepts, but it may also be useful to consider the negative vectors it rejects. Alternatively, it is possible to use the full learner to find partial preconditions. The full learner can be called with mixed groups included among the negative groups, just as the partial learner can. This allows Alive-Infer to produce a precondition before the positive and negative examples are fully separated, but does not bound the complexity of the formula. Another possibility is to call the full learner with a single positive vector (corresponding to the largest positive group) and all the negative and mixed vectors. This also produces a formula that

rejects all negative vectors, but only providing one positive vector allows the learner to find a simpler formula. However, there is still no bound on the complexity of the formula it finds, and our experiments show that the weighted partial learner often finds preconditions that accept more positive examples.

4.5.3 Safety condition learning

Both formula learners find preconditions that accept some or all positive examples and reject all negative examples. To allow the use of partially-defined predicates, Alive-Infer interprets a precondition that is unsafe for an example (*i. e.*, $B(\phi, e) = \star$) as rejecting that example. However, an Alive transformation that is unsafe for any example is considered incorrect, as it may have an ambiguous SMT encoding or may cause a crash when executing the corresponding C++ implementation. Because the learned preconditions only have unsafe behavior for negative examples, it is possible to avoid any unsafe behavior by adding additional clauses that reject unsafe examples before the remainder of the precondition is evaluated. (Recall that $\bot \wedge \star = \bot$.)

A safety prefix must reject all (negative) examples where the precondition is unsafe, but accept all positive examples. The prefix is free to accept or reject negative examples where the precondition is safe. The procedure EnsureSafety in Algorithm 4.5 reuses InferByExamples from Algorithm 4.2 to infer a prefix ψ for a precondition ϕ such that $\psi \wedge \phi$ is safe for all examples.

The arguments to EnsureSafety include the candidate precondition ϕ and the sets E^+, E^- of positive and negative examples, respectively. Line 2 finds the set $E^\star$ of *unsafe* negative examples. The prefix ψ is obtained by calling InferByExamples with E^+ and $E^\star$. Next, EnsureSafety verifies that ψ rejects all examples where ϕ is unsafe. Line 6 searches for examples where ψ does not guarantee the safety of ϕ, using the SMT query $\exists_{\mathcal{C}} : \psi \wedge \neg\sigma_p$. If any are found, they are added to the set of unsafe examples $E^\star$ and a new prefix is inferred. If none are found, EnsureSafety checks that ψ does not reject any examples that ϕ accepts. Line 10 searches for positive examples that ψ rejects, using the SMT query $\exists_{\mathcal{C}} : \sigma_p \wedge \phi \wedge \neg\psi$. Both queries are performed with respect to all feasible type assignments for T.

Algorithm 4.5 Add a prefix to a precondition that prevents unsafe behavior

```
1: procedure EnsureSafety(T, E+, E−, φ)
2:     E⋆ ← {e : e ∈ E−, B(φ, e) = ⋆}
3:     P ← ∅
4:     loop
5:         ψ, P ← InferByExamples(T, P, E+, E⋆)
6:         e⋆ ← {e : B(ψ ∧ φ, e) = ⋆}
7:         if e⋆ ≠ ∅ then
8:             E⋆ ← E⋆ ∪ e⋆
9:         else
10:            e+ ← {e : B(φ, e) = ⊤ ∧ B(ψ, e) = ⊥}
11:            if e+ = ∅ then
12:                return ψ ∧ φ
13:            else
14:                E+ ← E+ ∪ e+
15:            end if
16:        end if
17:    end loop
18: end procedure
```

The queries above implicitly assume that the prefix ψ is always safe. If ψ is unsafe for some examples, it is possible to repeat the inference process and find an additional prefix to make it safe. Fortunately, the safety conditions tend to be very simple in comparison to the precondition, so the chance that the prefix will itself have a nontrivial safety condition are negligible. In fact, the Alive-Infer prototype does not implement this loop, instead signalling an error if σ_ψ is nontrivial. This error has not yet been observed in testing.

4.6 Generalizing Concrete Transformations

Concrete program transformations produced by other tools are often instances of more general optimizations, but these generalized transformations may require preconditions in order to be correct. One source of concrete transformations is super-optimization [81], which takes a program or program fragment and finds the smallest program that has the same behavior (or refines the behavior). The Souper project [57] super-optimizes LLVM IR, by systematically searching for code fragments that can be simplified into single values. Figure 4.6(a) shows one such transformation.

```
%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
```

(a) A pattern found by Souper

```
Pre: 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
```

(b) Generalization of (a) with inferred precondition

Figure 4.6: A Souper-generated pattern (a) can be generalized by replacing fixed constants with symbolic constants. Alive-Infer can then find a precondition, as in (b).

Patterns found by Souper or other super-optimizers can be generalized by loosening type constraints and replacing specific constants with symbolic constants, with a precondition relating the values for the constants. This generalized transformation will be applicable in far more scenarios, and may even be worth adding to the peephole optimizer. Figure 4.6(b) shows a generalization of a Souper pattern with a precondition found by Alive-Infer. Surprisingly, the partial precondition used would not accept the original pattern: note that `C4` generalizes a zero, but the precondition requires it be non-zero. Nevertheless, the partial precondition accepts the majority of positive examples.

Alive-Infer also generates a full precondition, which accepts the original pattern:

```
(C3 != 0 || C2 == 0)
&& (C4 & ~C1 != 0 || C4 == C2)
&& (C3 != 0 || C1 == -C2)
```

While this precondition accepts all possible positive examples, the partial precondition is more succinct.

Generalizing a transformation with Alive-Infer is easily done. Once the transformation is expressed in Alive syntax, each literal constant in the source is replaced with a fresh symbolic constant.[3] If desired, explicit type requirements can also be removed.

[3]Generalizing constants in the target is more difficult, because the target cannot introduce new variables. Explicit type constraints can also be removed. It is possible to cheat by introducing no-op instructions to the source that introduce new symbolic constants, such as `select 1, %6, C5`, but a

Alive-Infer will then learn preconditions that restrict the possible values for the new symbolic constants, producing a correct, generalized transformation.

4.7 Evaluation

The Alive-Infer prototype is an extension of Alive-NJ that provides precondition inference and some additional syntax for specifying assumptions and providing initial preconditions.

When operating with a transformation that already has a precondition, the prototype will report how many positive and negative examples it correctly accepts and rejects, respectively. Optionally, the Alive-Infer can extract predicates from the precondition to use as the initial predicate set.

4.7.1 Effectiveness of Alive-Infer

To test the effectiveness of the inference algorithm, Alive-Infer was used to re-infer preconditions for transformations that were derived from InstCombine during the development of Alive. Of the 415 transformations, 195 do not require preconditions, 41 rely on dataflow analyses, and seven require predicates or constant functions that were not supported by Alive-Infer at the time. This leaves 174 transformations for which Alive-Infer could be expected to find a non-trivial precondition.

Experiments were performed on a computer with a four-core, 64-bit Intel Skylake processor and 16 GiB of RAM. SMT queries were solved using Z3 4.4.1 [28]. Alive-Infer was configured to request, for each selected type model, 10 positive and 10 negative examples from the solver and to randomly generate 500 additional examples, discarding duplicates. To minimize the influence of solver time, test for finding full and partial preconditions were performed separately, and safety prefix learning was not enabled. When finding full preconditions, Alive-Infer was configured to not generate partial

---

more permanent solution would be needed if this technique became popular. Note that such a method would find a relation between the source and target constants. This would need to be reduced to a function before the transformation could be used.

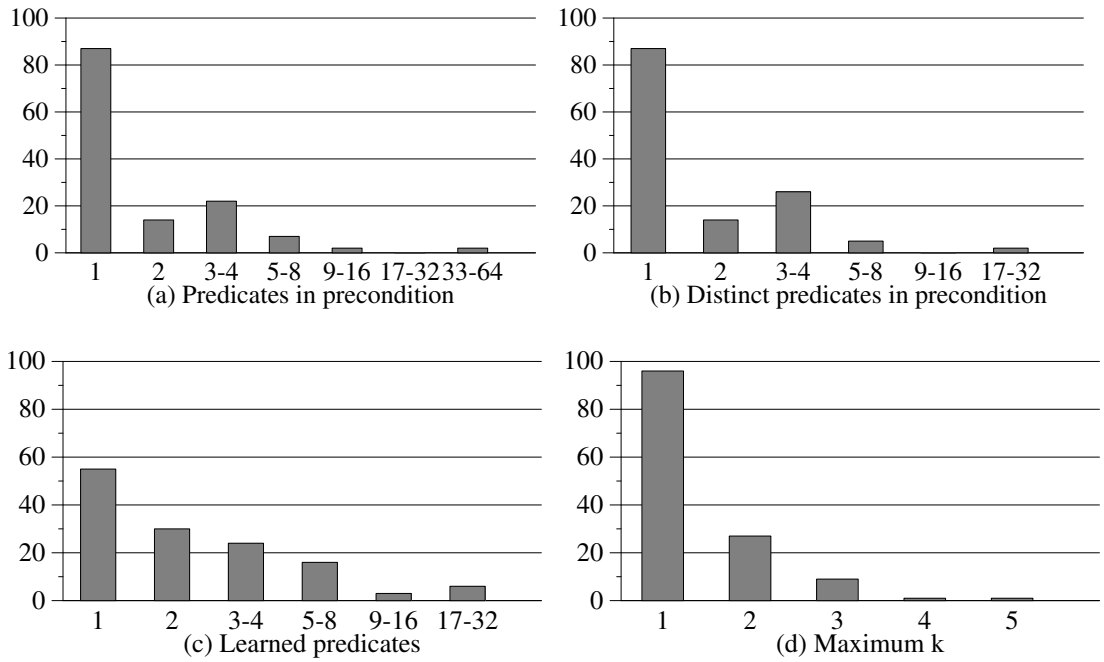


Figure 4.7: Information about the full preconditions found within 1000 seconds. The histograms group preconditions based on (a) the number of atomic predicates, (b) the number of distinct predicates used, (c) the number of predicates learned during inference, and (d) the maximum size of a disjunctive clause in the predicate (*i. e.*, the value of k needed by the formula learner).

preconditions. When finding partial preconditions, Alive-Infer was configured to halt after finding one sufficient precondition.

Inference within a time limit

For 133 of the 174 transformations, Alive-Infer found a full precondition within 1000 seconds. For an additional 31 transformations, Alive-Infer found a partial precondition within 1000 seconds.

Figure 4.7 gives some information about the full preconditions that were found, including the number of atomic predicates in the formula, the number of distinct predicates used, the number of predicates learned during the inference process, and the maximum disjunction size used during formula learning. As Figure 4.7(a) indicates, about 80 transformations required only a single atomic predicate, with an additional

40 requiring two, three, or four. Comparison with Figure 4.7(b) shows that many preconditions included multiple uses of the same atomic predicate (or its negation). This is common in CNF formulae (*e. g..*, $a \vee (b \wedge c)$ must be expressed as $(a \vee b) \wedge (a \vee c)$).

Comparison of Figures 4.7(b) and 4.7(c) shows that Alive-Infer often learned predicates that were not needed by the formula learner, but did not learn more than 32 predicates before separating the positive and negative examples. Figure 4.7(d) shows that 45 preconditions involved at least one disjunction (*i. e.*, $k > 1$).

Comparison with initial preconditions

As all of the 174 transformations tested already had preconditions, it is possible to compare those preconditions (ϕ_0) with the preconditions found by Alive-Infer (ϕ_1). If it is possible to satisfy $\phi_1 \wedge \neg\phi_0$ for some feasible type assignment, then Alive-Infer has found a weaker precondition. Of the 133 full preconditions found by Alive-Infer during testing, 73 were weaker (meaning the precondition derived from LLVM was partial). Occasionally, this is due to additional requirements needed to avoid compiler non-termination (see Chapter 5) or, more often, because the full precondition accepts edge cases that cannot occur due to interaction with other transformations.

As an example where the precondition found by Alive-Infer is a considerable improvement over the precondition used by LLVM, consider this example of a bit test simplification, one of twelve variants of a single general transformation in LLVM's InstructionSimplify pass:

```
Pre: C1 == ~C2
  %l = and %X, C1
  %c = icmp eq %l, 0
  %t = and %X, C2
  %r = select %c, %t, %X
=>
  %r = %X
```

Alive-Infer infers the precondition `~C1 & ~C2 == 0`, which accepts a strict superset of examples. All twelve variants remain correct when using this weaker precondition.

4.7.2 Finding preconditions through enumeration

To assess the value of the two-part strategy of learning predicates and then a formula, we created a variant of the Alive-Infer prototype that instead finds preconditions through enumeration.

The modified prototype uses an extended predicate enumerator that includes compound predicates using $\wedge$, $\vee$, and $\neg$. Each enumerated predicate is tested to see whether it accepts all positive examples and rejects all negative examples. If so, it is then verified using the tests from Algorithm 4.1, which search for additional positive and negative examples that the predicate rejects or accepts, respectively. If no additional examples exist, the predicate is proposed as a full precondition. If no additional negative examples are found, it is proposed as a partial precondition.

This strategy, which we call *Alive-Search* to distinguish it from Alive-Infer, is guaranteed to find a minimally-sized sufficient precondition, because the predicates will be enumerated in nondecreasing order of size. Unlike Alive-Infer, it cannot get stuck attempting to learn a complex atomic predicate that can be expressed more succinctly as a combination of smaller predicates. On the other hand, Alive-Search cannot divide the search problem into smaller parts. To find a precondition with two predicates of sizes m and n, Alive-Search will need to consider $O(c^{m+n})$ preconditions. In contrast, Alive-Infer need only consider $O(c^m + c^n)$ predicates if it chooses samples appropriately, which is a vast improvement in the best case. The number of preconditions or predicates to consider grows so quickly with size that Alive-Infer can spend the majority of its time learning predicates that will be discarded later and still find a result faster than Alive-Search.

Because Alive-Search does not divide the examples to create subproblems, it is not guided by its examples in the same way as Alive-Infer. Instead, the examples serve as a filter, determining which preconditions will be subject to full verification. Despite the initial overhead of generating examples, this speeds up the search overall, as checking that a precondition accepts or rejects an example is faster than using a solver to verify a transformation's correctness.

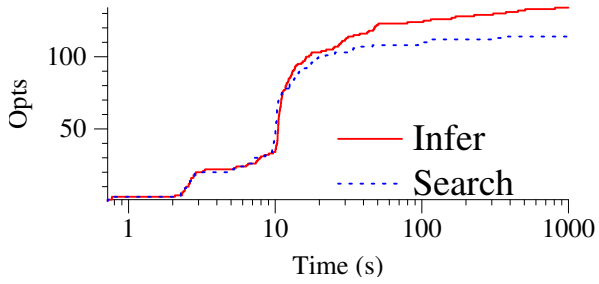


Figure 4.8: Cumulative number of preconditions that were found under a time limit. The x axis is a time limit, and the y axis is the number of tests that found a precondition for a transformation in that time or less. The jump between two and three seconds is caused by example generation.

Alive-Search was tested on the same 174 transformations as Alive-Infer, using the same number of examples. It was able to find a full precondition within 1000 seconds for 114 transformations. Recall that Alive-Infer found a full precondition within 1000 seconds for 133 transformations.

Figure 4.8 shows the number of transformations for which Alive-Infer and Alive-Search found a full precondition within a time limit. Given a time limit of ten seconds, both methods are able to find full preconditions for about 100 transformations. These were typically preconditions that required only one or two atomic predicates. For more complex preconditions, Alive-Infer is more likely to find a precondition within a given time limit than Alive-Search.

4.7.3 Generalizing concrete transformations

To judge the usefulness of this technique, we generalized 71 concrete transformations generated by Souper, generalized their constants, and attempted to infer preconditions. Alive-Infer found full preconditions for 51 transformations, and partial preconditions for an additional three transformations.

The remaining 17 cases expose some limitations to the Alive-Infer approach. Alive-Infer, like Alive itself, relies on an SMT solver in several key places. Solvers such as Z3 do very well for many queries, but it is still possible to formulate queries that Z3 cannot efficiently resolve. Certain combinations of instructions, particularly multiplication and division, can produce unexpectedly challenging correctness conditions and unresolvable

queries. Perhaps because of the way they are generated, these difficult queries seem to occur more frequently when generalizing Souper patterns than when working with LLVM-derived transformations. Inferring and verifying these difficult transformations may become more feasible in the future, as SMT solvers continue to improve and as the Alive toolkits are updated to use more specialized solving techniques.

4.8 Summary

Alive-Infer demonstrates that data-driven precondition inference for Alive transformations is feasible. Given a transformation, Alive-Infer can generate and classify examples automatically. Guided by the examples, Alive-Infer finds a set of useful predicates that are generalized over types and need not be defined for all examples. These predicates are used to create preconditions that are sufficient to make the transformation correct, with some intended to accept as many examples as possible and others designed for brevity. The Alive-Infer prototype has successfully found preconditions for many Alive transformations, drawn from LLVM's peephole optimizer and from super-optimizer patterns, showing that it is applicable to real-world transformations.

Chapter 5

Detecting Non-Termination

Alive and Alive-Infer help developers write peephole transformations that are correct, meaning they do not introduce new behavior into a program, but there are other classes of compiler bugs. Some may cause the compiler to crash or enter an infinite loop: instead of producing an incorrect executable program, the compiler produces no executable program. The safety conditions in Alive (see Section 3.2.2) protect against some compiler crashes, but other bugs cannot be detected by analyzing transformations in isolation. One class of bugs occurs when applying a transformation or sequence of transformations creates new opportunities to apply it. In such cases, the repeated application of the transformation can apply indefinitely, resulting in compiler non-termination.

The InstCombine pass in LLVM operates by maintaining a set of unexamined instructions, which initially contains all instructions. At each step, it removes an instruction from that set and determines whether any of its transformations apply to it. If so, it applies the transformation, adding any new instructions created by the transformation to the unexamined set. The process continues until all instructions have been examined. This ensures that all opportunities to apply transformations are found, even if those opportunities are created by other transformations. However, it is vulnerable to non-termination, because each iteration removes one instruction but may add one or more new instructions.

Figure 5.1 shows a pair of transformations, each of which can undo the work of the other. Notably, they are not simply inverses: not all output from one transformation can be transformed by the other. Additionally, both are desirable transformations: one potentially reduces the number of set bits in a constant argument to **xor**, possibly allowing the **xor** to be removed entirely if its second argument becomes zero. The other

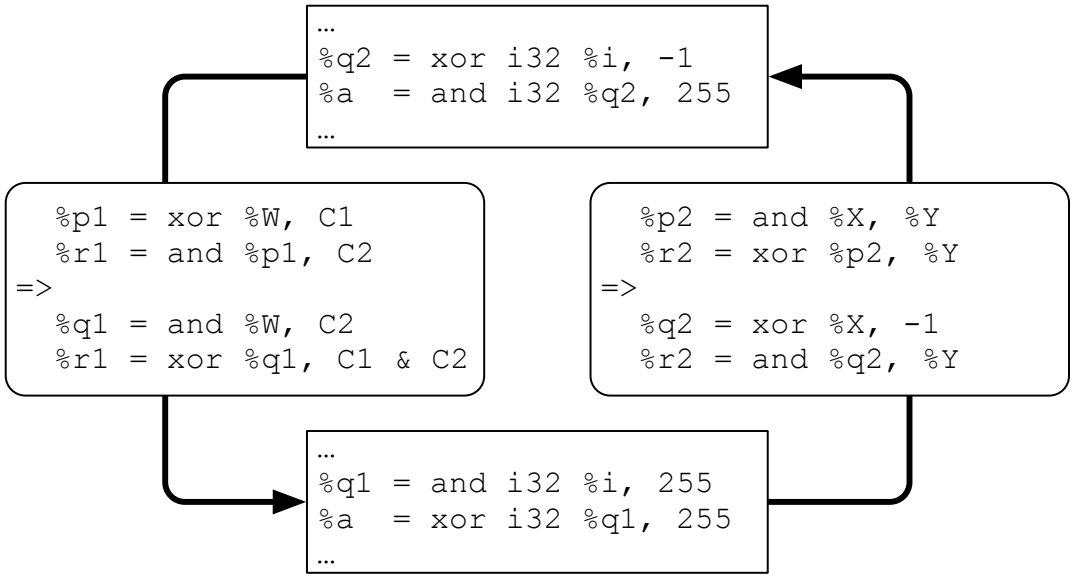


Figure 5.1: Two correct transformations that together cause non-termination, replacing the upper fragment with the lower and vice versa indefinitely. Both transformations are desirable: the left one reduces the number of set bits in the second argument to **xor**, the right one reduces the number of references to `%Y`.

reduces the number of references to a value `%Y`, simplifying later compilation. Thus, preventing the infinite loop by removing one or the other transformation is undesirable.

There are several ways to ensure termination. One simple possibility is to cap the number of instructions that may be examined. Once a threshold number of transformation applications is reached, optimization halts and any remaining unexamined instructions are left unexamined. This strategy is unsatisfying for two reasons. If the threshold is low, many programs may be left only partially optimized. If the threshold is high, any program that triggers non-termination will waste time repeatedly transforming the same section of a program, possibly leaving other sections unexamined.

A less simple method for ensuring termination is to compare the program before and after each transformation to see whether any changes have been made. Unfortunately, this approach has several problems. First, any such comparison must be incomplete, since it is undecidable whether two arbitrary programs have the same behavior. Second, the transformation process may be cycling through an unknown number of intermediate programs. To reliably detect this, InstCombine would need to remember every previous

form of the program and compare the program with all of its predecessors after every transformation. This is an enormous amount of work to detect a situation that ideally should never occur—and even this does not actually guarantee termination. While such transformations are unlikely to be added to a compiler, it is easy to create transformations that will produce an infinite sequence of distinct intermediate programs (*e. g.*, replacing $x - y$ with $0 - (y - x)$).

Both methods require checks during compilation to ensure termination. These checks could be avoided if the peephole optimizer as a whole were known to always terminate. InstCombine rewrites code into a broadly-defined canonical form, and uses preconditions to avoid applying certain transformations when doing so is not profitable. Transformations may make code smaller, or reduce the number times a result is used, or simply enable further transformation. For example, both transformations in Figure 5.1 improve code in certain circumstances, but the loop depicted is avoided by adding the precondition `C2 != C1 & C2` to the left transformation, ensuring it does not create an opportunity for the right transformation.

A formal termination analysis of InstCombine would require formalizing this canonicalization property, and then showing that each transformation it defines is profitable. Automating such an analysis of the C++ implementation seems impractical, even if the nature of profitability can be formally defined.

This chapter describes a method for detecting non-termination that leverages the high-level view of transformations provided by Alive. Given a set of transformations specified in Alive, it searches for sequences that might give rise to non-termination. For a sequence of transformations, it determines whether a program exists such that the sequence can be performed indefinitely. If such a program does exist, it can generate a concrete LLVM IR program to demonstrate the non-termination. The method relies on *transformation composition*, a process for deriving a single transformation that has the effect of applying two or more transformations to an input program sequentially. Using composition, it compares the requirements for applying a (possibly composite) transformation to the requirements for applying it twice. If they are the same, then the transformation can be applied arbitrarily many times.

```
Pre: isPowerOf2(C+1)
  %r = mul %X, C
=>
  %y = shl %X, log2(C+1)
  %r = sub %y, %X
```

(a) Proposed transformation

```
  %y = shl %X, C
  %r = add %y, %X
=>
  %r = mul %X, (1<<C)+1
```

(b) Existing transformation

Figure 5.2: A developer testing transformation (a) in LLVM discovered that it triggered an infinite loop [47]. The new transformation reversed the work of an already-existing transformation (b), preventing InstCombine from reaching a steady state.

This method can be used to analyze existing sets of transformations, and can also be used to determine whether new transformations may cause non-termination if added to InstCombine. The guidelines for whether a transformation is profitable are informal, and it is easy for a developer to accidentally introduce a new transformation that works against an existing transformation. For example, a developer working to resolve a bug in LLVM added the transformation shown in Figure 5.2(a). It replaces a multiply with a shift and add, which can be performed more efficiently on some hardware. This addition introduced an infinite loop into LLVM [47], which was eventually found to be caused by interaction with the pre-existing transformation shown in Figure 5.2(b). The earlier transformation reduces the number of instructions in a program, so its results are considered more canonical for InstCombine. The new transformation was instead added to the back-end of LLVM, to be used only on architectures where it was appropriate.

The method for detecting non-termination is given in three parts. Section 5.1 describes the process of transformation composition, Section 5.2 shows how to determine whether a given sequence of transformations can be applied indefinitely, and Section 5.3 gives a method for generating such sequences. To confirm that non-termination occurs, Section 5.4 describes how to generate a concrete input that will trigger a sequence of transformations. Finally, Section 5.5 describes the experiments used to evaluate this method.

5.1 Composing Transformations

Transformation composition takes two Alive transformations and creates a single transformation that has the effect of applying the original two transformations in sequence. Figure 5.3(c) shows the result of composing the two transformations from Figure 5.1.

The process is called "composition" by analogy to function composition, although transformation composition is different in several key respects. Two transformations may not compose, or may compose in multiple ways. Three or more transformations may compose differently depending on the order in which the compositions are performed (*i. e.*, composition is not associative).

To compose two transformations A and B, we must determine whether it is possible to apply B to a program resulting from an application of A. In a sense, this is always possible, because the code fragments transformed by A and B might not overlap, but we disregard those trivial cases. For our purposes, we specifically are interested in cases where A enables B, meaning B transforms an instruction or constant that was created by A.

For brevity, we will write A_s, A_t, and A_p to indicate the source, target, and precondition of a transformation, respectively. The sub-DAG A_t represents code that is guaranteed to be present after A has been performed. Algorithm 5.1 simulates the matching process of B, attempting to find an arrangement whereby B_s will match A_t. Instructions only match if their opcodes are the same, but input variables can match any value.

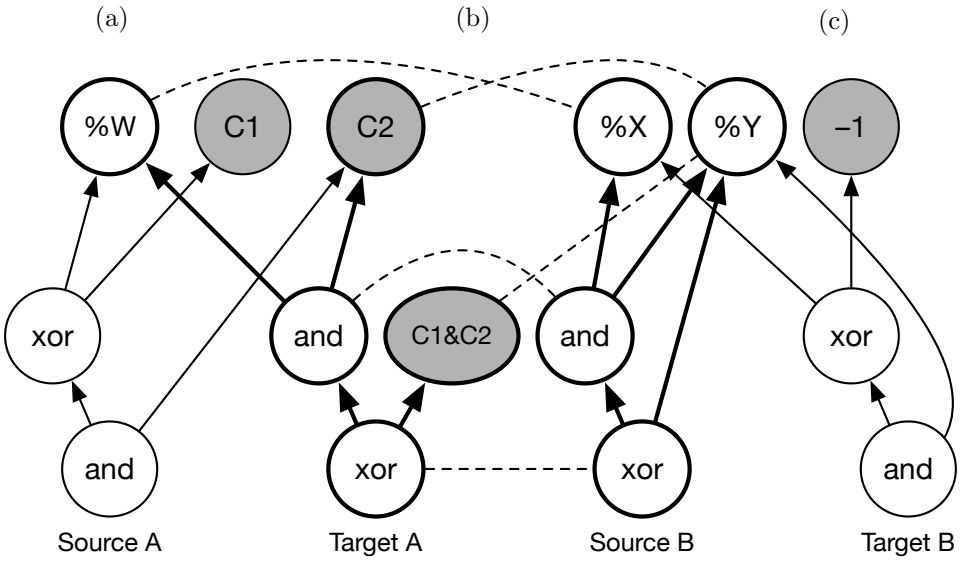
Algorithm 5.1 has three major parts. First, it determines whether A_t and B_s have the same shape, meaning each instruction in one corresponds to a similar instruction in the other or an input variable. The procedure AlignDAGs, described in Section 5.1.1, compares the shapes and groups the terms in A_t and B_s into sets. Each set contains terms that must be equal in order for B to apply after A. For example, a set might contain **and** instructions from A_t and B_s.

In Figure 5.3(d), alignment begins at the **xor** nodes labeled "Target A" and "Source B", and then proceeds through their descendents, indicated with thick node outlines

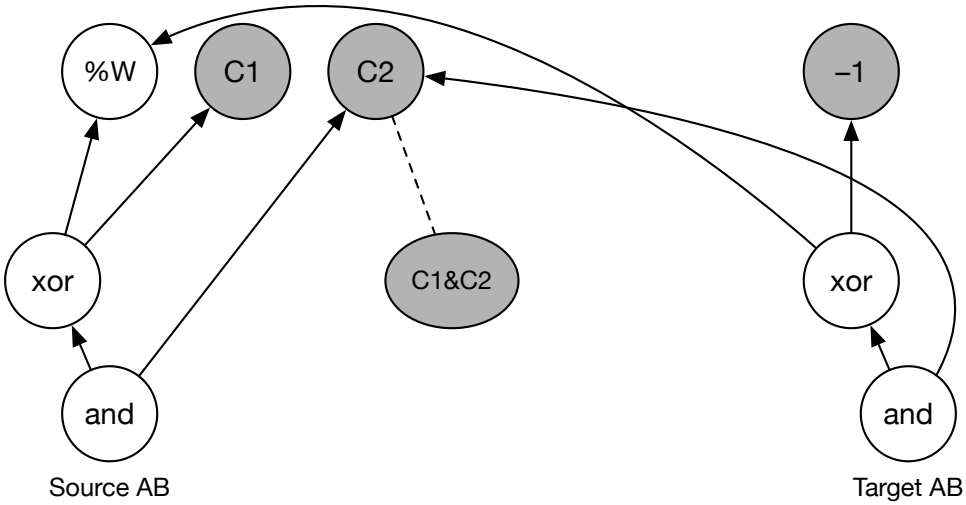
```
Name: A
  %p1 = xor %W, C1
  %r1 = and %p1, C2
=>
  %q1 = and %W, C2
  %r1 = xor %q1, C1 & C2
```

```
Name: B
  %p2 = and %X, %Y
  %r2 = xor %p2, %Y
=>
  %q2 = xor %X, -1
  %r2 = and %q2, %Y
```

```
Name: AB
Pre: C2 == C1 & C2
  %p1 = xor %X
  %r1 = and %p1, C2
=>
  %q2 = xor %X, -1
  %r1 = and %q2, C2
```



(d) DAG representations of A and B. Alignment begins with the target root of A and source root of B, indicated with thick outlines. Dashed lines connect nodes that match during alignment.



(e) DAG representation of AB, constructed from the source of A and the target of B. Nodes connected by dashed lines in (d) have been replaced with a representative (*e. g.*, `%Y` is replaced by `C2`). The dashed line connecting `C2` and `C1 & C2` indicates that they both matched `%Y` and must therefore be equal.

Figure 5.3: Composition of two transformations. Application of AB is equivalent to applying A followed by B.

Algorithm 5.1 Create a transformation AB that composes A and B

```
1:  procedure Compose(A, B)
2:      Sets ← AlignDAGs(A_t, B_s)
3:      CheckValidity(Sets)

4:      φ ← ⊤
5:      for S ∈ Sets do
6:          p, Sets ← SelectReplacement(S, Sets)
7:          φ ← φ ∧ p
8:      end for

9:      AB_s ← Graft(A_s, Sets)
10:     AB_t ← Graft(B_t, Sets)
11:     AB_p ← Graft(A_p, Sets) ∧ Graft(B_p, Sets) ∧ φ
12:     return AB
13: end procedure
```

and arrows. Dashed lines connect nodes that must be equal in order for the DAGs to align.

Alignment may fail under certain circumstances, such as attempting to match instructions with different opcodes, but other requirements are easier to check after AlignDAGs completes. the procedure CheckValdity, described in Section 5.1.2, looks for certain impossible situations, such as an instruction being equal to a constant, and reports that composition was impossible in those circumstances.

If alignment succeeds, the second part of Compose chooses a distinguished value for each set of equal terms that will be used when constructing the composite transformation. The procedure SelectReplacement, described in Section 5.1.3, generally chooses the most specific term in each set (*e. g.*, an instruction is more specific than an input variable) and notes it for later. Some sets, such as those containing multiple constant expressions, require additional preconditions in order to be equal. These conditions are returned by SelectReplacement and will be incorporated into the final composite.

Once replacements have been selected, the third part of Compose creates the composite transformation AB by creating copies of A_s (*i. e.*, the input program before applying A) and B_t (*i. e.*, the input program after applying B), but replacing any terms that were unified with the selected replacement for its set. For example, an input

variable in A that matched an instruction in B_s will be replaced by that instruction in AB_s. Section 5.1.4 describes this process. In Figure 5.3(e), the composite AB is built from the sub-DAGs rooted at "Source A" and "Target B" in Figure 5.3(d), with nodes that participated in alignment replaced by their selected replacement. For example, the second argument to the **and** in B_t is `%Y`, which unified with `C2` and `C1&C2`. Of these, `C2` is selected as the replacement, and becomes the second argument to the **and** in AB_t. The required equality between `C2` and `C1&C2` will become part of the precondition for AB.

Algorithm 5.1 describes the simplest case of composition, which aligns the roots of A_t and B_s. The algorithm for composition where one of the roots aligns with a non-root term is similar. The differences and additional details are described in Section 5.1.5.

5.1.1 DAG alignment

The procedure AlignDAGs(C, P) determines whether the sub-DAGs C and P can be given the same shape by replacing input variables. The process is designed to simulate the matching portion of an Alive transformation, which tests whether a transformation can apply to an IR code fragment. The argument C (for "code") corresponds to the program being transformed. The argument P (for "pattern") corresponds to the source pattern of an Alive transformation. If successful, it returns the terms of C and P organized into disjoint sets, with each set containing terms that must unify in order for C and P to align.

The alignment process begins with the roots of C and P and proceeds recursively through their subterms. Two terms align only if (1) either or both are input variables, (2) both are instructions with the same opcode and their corresponding arguments align, or (3) both are constants (including symbolic constants and constant expressions). Algorithm 5.2 gives an implementation of AlignDAGs that tests alignment and collects the sets of unified terms.

The algorithm maintains two data structures: *Sets* keeps track of the sets of unified terms, and *worklist* contains pairs of terms that must unify in order for alignment to succeed. The former requires the ability to determine which set a term belongs to and

Algorithm 5.2 Determine whether DAG C is matched by DAG P and return sets of unified terms

```
 1: procedure AlignDAGs(C, P)
 2:     Sets ← MakeCodeSets(C) ∪ MakePatternSets(P)
 3:     worklist ← {⟨C, P⟩}
 4:     while worklist ≠ ∅ do
 5:         ⟨t1, t2⟩, worklist ← pop(worklist)
 6:         S1, p1, c1 ← Lookup(t1, Sets)
 7:         S2, p2, c2 ← Lookup(t2, Sets)
 8:         if S1 ≠ S2 then
 9:             S, Sets ← Unify(S1, S2, Sets)
10:             if c1 ≠ ⊥ ∧ c2 ≠ ⊥ then
11:                 fail
12:             else if c1 ≠ ⊥ ∧ p2 ≠ ⊥ then
13:                 pairs ← Match(c1, p2)
14:                 worklist ← worklist ∪ pairs
15:             else if p1 ≠ ⊥ ∧ c2 ≠ ⊥ then
16:                 pairs ← Match(c2, p1)
17:                 worklist ← worklist ∪ pairs
18:             else if p1 ≠ ⊥ ∧ p2 ≠ ⊥ then
19:                 ⟨p, pairs⟩ ← Merge(p1, p2)
20:                 Sets ← DesignatePattern(S, p, Sets)
21:                 worklist ← worklist ∪ pairs
22:             end if
23:         end if
24:     end while
25:     return Sets
26: end procedure
```

to unify two sets. The prototype uses the disjoint sets structure from the union-find algorithm. In addition to tracking set membership, *Sets* also associates an optional *code instruction* or *pattern instruction* for a set. These are used when bringing code into alignment. Initially, each subterm in C and P is placed into its own set, with instructions in C and P marked as code and pattern instructions, respectively. The *worklist* initially contains the pair $\langle C, P \rangle$.

Next, AlignDAGs draws pairs from *worklist* and checks whether they can be brought into alignment. It looks up the sets for the terms (along with their code and pattern instructions, if any). If the two terms belong to the same set, they already are aligned. Otherwise, it unifies the two sets and then checks whether they had contained any code or pattern instructions whose subterms need to be aligned.

Here the difference between code and pattern becomes important. If both sets have associated code instructions, alignment fails. This can occur when input variables are repeated, but in LLVM a repeated variable is tested using pointer equality (see Section 3.4.2). Distinct code instructions would have been created separately, meaning they would not be considered equal, if if they have the same opcode and arguments.

If one of the sets has an associated code instruction and the other has an associated pattern instruction, AlignDAGs calls Match(c, p) to test whether they can be aligned. This procedure checks whether c and p have the same opcode. If the opcode has an additional parameter, such as a comparison relation, it checks whether those are equal. If the instructions can have attributes, it makes sure the attributes of p are a subset of those in c. If all these tests succeed, it returns pairs containing corresponding arguments of c and p. Otherwise, alignment (and composition) fails.

If neither set has an associated code instruction, but both have associated pattern instructions, AlignDAGs calls Merge(p_1, p_2) to test whether they can be aligned. This is similar to Match, except for how it handles attributes. If p_1 and p_2 have different attributes, it creates a new instruction using the union of their attributes. This new instruction becomes the associated pattern instruction for the unified set. Otherwise, it returns p_1 or p_2 arbitrarily.

Alignment succeeds once no more pairs can be drawn from *worklist*.

5.1.2 Checking Validity

Several conditions must hold for one transformation to follow another. Some of these conditions are guaranteed by AlignDAGs, which can easily detect certain violations, such as matching instructions with different opcodes. Other conditions are more easily checked once alignment has completed. The procedure CheckValidity tests whether the alignment found by AlignDAGs corresponds to a possible pattern match. The necessary conditions are:

1. No set may contain unequal literal constants.
2. No set may contain both a constant and an instruction.
3. No set may contain both an instruction created in A_t and a value present in A_s. This would imply that A somehow matched an instruction that it itself created.
4. No set may depend on itself.

The first three conditions can be checked individually for each set, but the fourth condition is more complex. A term, such as an instruction, depends on its subterms. If alignment is successful, then the terms in a set are the same, and therefore depend on the subterm for every term in the set. Normally, a value in LLVM IR never depends on itself,[1] but the alignment process can introduce a self-dependency if terms are referenced multiple times. Figure 5.4 shows two transformations that have a circular dependency after alignment.

An alignment-introduced circular dependence exists if and only if the graph produced by taking the sub-DAGs for A_t and B_s and merging all unified nodes is cyclic. In Figure 5.4(c), this would merge the **add** with `%Y` and the **sext** with `%X`, resulting in a cycle between the merged nodes. It is not necessary to create this graph: it can be simulated by performing a graph traversal on the sets of unified terms. An arc from one set to another exists if a term in one set depends on a term in another. If a back edge is discovered when performing a depth-first traversal of this graph, then the graph has a cycle and the alignment has introduced a circular dependency.

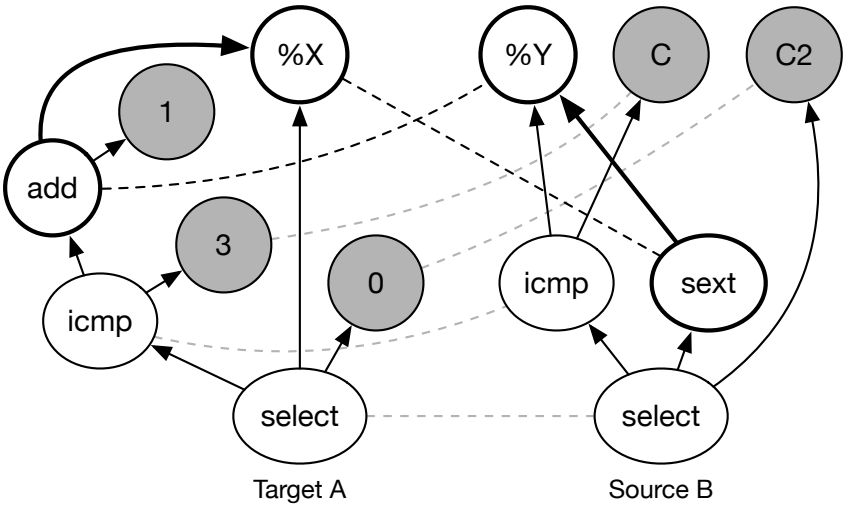
[1] Alive does not include `phi` nodes, which allow for indirect self-dependence.

```
Name: A
  %r   = sdiv 1, %X
=>
  %inc = add %X, 1
  %c   = icmp ult %inc, 3
  %r   = select %c, %X, 0
```

(a)

```
Name: B
Pre: C2 == sext(C-1) && C != 0
  %c  = icmp ult %Y, C
  %y  = sext %Y
  %r  = select %c, %y, C2
=>
  %c2 = icmp ugt %y, C2
  %r  = select %c2, C2, %y
```

(b)



(c) Sub-DAGs A_t and B_s, with dashed lines connecting unified nodes

Figure 5.4: Alignment failure due to circular dependency. In (c), note that `%X` is equal to the **sext** in B_s, which depends on `%Y`, which is equal to the **add** in A_t, which depends on `%X`. Such a circular dependence could not occur in LLVM IR.

5.1.3 Selecting Replacements

Once AlignDAGs and CheckValidity have determined that composition is possible and found the sets of unified terms, SelectReplacements chooses a single term to represent each set. Additionally, it returns a predicate that must hold in order for composition to succeed.

To choose, SelectReplacements($S, Sets$) orders the terms in S by priority. In decreasing order, the priorities are:

1. The code or pattern instruction obtained from alignment
2. A literal constant
3. A constant expression
4. A symbolic constant
5. An input variable

If multiple terms have the same priority, one is chosen arbitrarily.

Under most circumstances, the predicate returned by SelectReplacements is trivial. If a set contains multiple constant expressions, SelectReplacements returns a predicate requiring them to be equal. In addition, if a set contains a symbolic constant or input variable from A and its replacement is a constant expression, SelectReplacements instead chooses a symbolic constant as the replacement, in order to prevent a constant expression from appearing in the source of the composite transformation. If the set has no symbolic constant, a fresh one is created. The predicate requires the selected symbolic constant be equal to the constant expression. For example, in Figure 5.3 the set containing `C2`, `C1&C2`, and `%Y` gets `C2` as its replacement and the predicate `C2 == C1 & C2` is added to the precondition.

5.1.4 Constructing the composed transformation

If A_t and B_s align, and we have obtained the sets of equal values and chosen replacements, we can construct the composed optimization. Algorithm 5.3 gives the procedure

GRAFT, which creates a copy of a term t, recursively substituting subterms according to the replacements selected by SELECTREPLACEMENT. If t has a replacement other than itself, GRAFT calls itself for the replacement and returns that result. This recursion is guaranteed to terminate, as CHECKVALIDITY has prevented any circularity in the replacement process. If t does not have a different replacement, GRAFT creates a copy of t, calling itself recursively for each argument, and returning the modified copy.

Algorithm 5.3 Clone a DAG, replacing nodes belonging to a set in $Sets$

```
1: procedure GRAFT(t, Sets)
2:     if ∃_{S∈Sets} : t ∈ S ∧ t ≠ REPLACEMENT(S, Sets) then
3:         return GRAFT(REPLACEMENT(S, Sets), Sets)
4:     else
5:         r ← COPY(t)
6:         for all parameters i do
7:             r_i ← GRAFT(t_i, Sets)
8:         end for
9:         return r
10:    end if
11: end procedure
```

In the context of COMPOSE, GRAFT is used to rewrite A_s, B_t, A_p, and B_p. The first two become the source and target of the composed transformation, and the latter two become part of the precondition.

It is possible that the composed precondition will refer to instructions that were created by A and then discarded by B and therefore do not occur in the source or target of the composed transformation. This is not normally allowed for Alive transformations, but is acceptable when the composed transformation is used for detecting non-termination.

Figure 5.5 shows an example of grafting where some input variables have unified with instructions. The result of alignment is shown in Figure 5.5(c): the terms in A_t and B_s that were examined have thick outlines, and dashed lines connect terms that unified. Note that the variables `%Z1` and `%X2` have unified with **add** instructions in B_s and A_t, respectively, and the roots have unified with each other. In the first two cases, the **add** will be chosen as the replacement. In the latter, the choice is arbitrary, but also irrelevant, as the roots will not be included in the composite AB.

Figure 5.5(d) shows AB, built by using GRAFT to rewrite A_s and B_t. The copies of terms that depended on `%Z1` or `%X2` instead depend on their replacements.

5.1.5 Off-root composition

Algorithm 5.1 describes the simple case of composition, where the root of A_t and B_s align. In the more complex cases, one of the roots aligns with a subterm. In these cases, the source or target of the composite transformation is constructed from the sources or targets of both transformations.

In the case where the root of A_t matches a non-root value in B_s, there will be a portion of B_s that does not align with any part of A_t. These subterms were not matched by A and so were unchanged when B applied. Thus, AB_s will include a portion of B_s, but with the term that matched with A_t replaced by A_s, in order to describe the input program before A or B applied.

Algorithm 5.4 shows how to compose the root of A_t to a non-root node in B_s. The first two sections of COMPOSEROOTTONONROOT are similar to COMPOSE, except that b is passed to ALIGNDAGS instead of B_s. Once replacements are selected, A_s becomes the replacement for the set containing A_t and b. Thus, when B_s is rewritten, it will replace b with A_s.

Figure 5.6 shows an example of composing a root to a non-root term, using the same self-composition as Figure 5.5. In Figure 5.6(c), the initial alignment is the **add** terms `%c1` (in the first target) and `%b2`. Alignment proceeds with their subterms, and replacements are chosen as usual, except that the set containing `%c1` (target) and `%b2` will be replaced by `%c1` (source). The composite transformation in Figure 5.6(d) is created by rewriting the source and target of the second copy of the transformation.

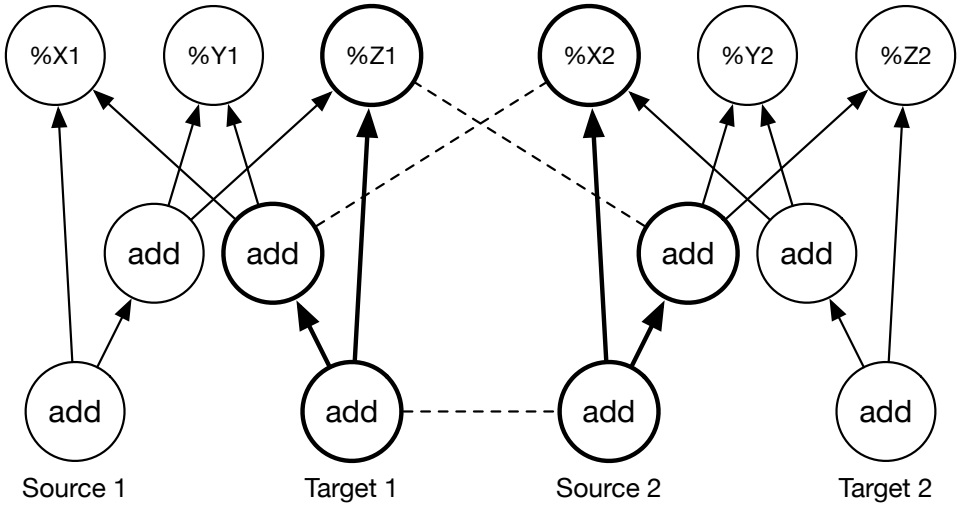
The converse case, where the root of B_s aligns with a subterm a of A_t is symmetric. Now, a is aligned with B_s, and B_t is designated as the replacement for the set containing a and B_s. Figure 5.7 shows the same self-composition as Figures 5.5 and 5.6, but now aligning the **add** terms `%a1` and `%c2` (in the second source). Alignment proceeds with their subterms and replacements are chosen as usual, except that the set containing `%a1` and `%c2` (source) will be replaced by `%c2` (target). The composite transformation

```
  %b = add %Y, %Z
  %c = add %X, %b
=>
  %a = add %X, %Y
  %c = add %a, %Z
```

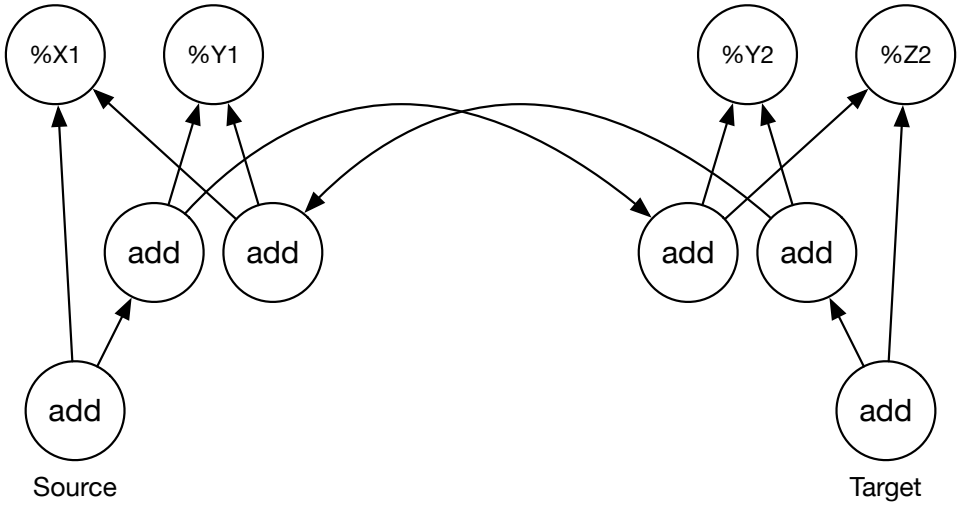
(a) A transformation that reassociates addition

```
  %b2 = add %Y2, %Z2
  %b1 = add %Y1, %b2
  %c1 = add %X1, %b1
=>
  %a1 = add %X1, %Y1
  %a2 = add %a1, %Y2
  %c1 = add %a2, %Z2
```

(b) Composition by matching `%c1` and `%c2`



(c) The DAGs for two copies of the transformation. Alignment begins with the root of the first target and the root of the second source, indicated by thick outlines. Dashed lines connect nodes put into the same set during alignment.



(d) The composed transformation, constructed from the first source and second target.

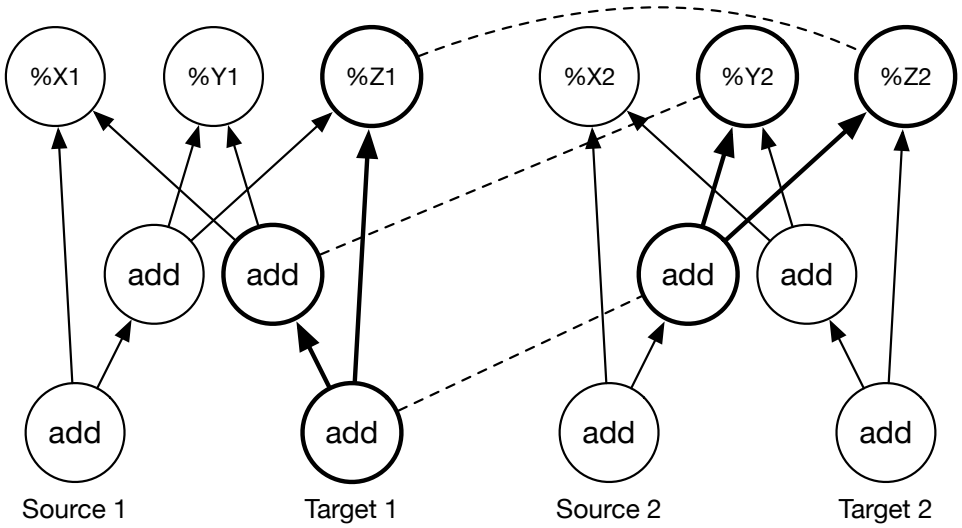
Figure 5.5: Composition of a transformation with itself, matching the roots of the source and target

```
  %b = add %Y, %Z
  %c = add %X, %b
=>
  %a = add %X, %Y
  %c = add %a, %Z
```

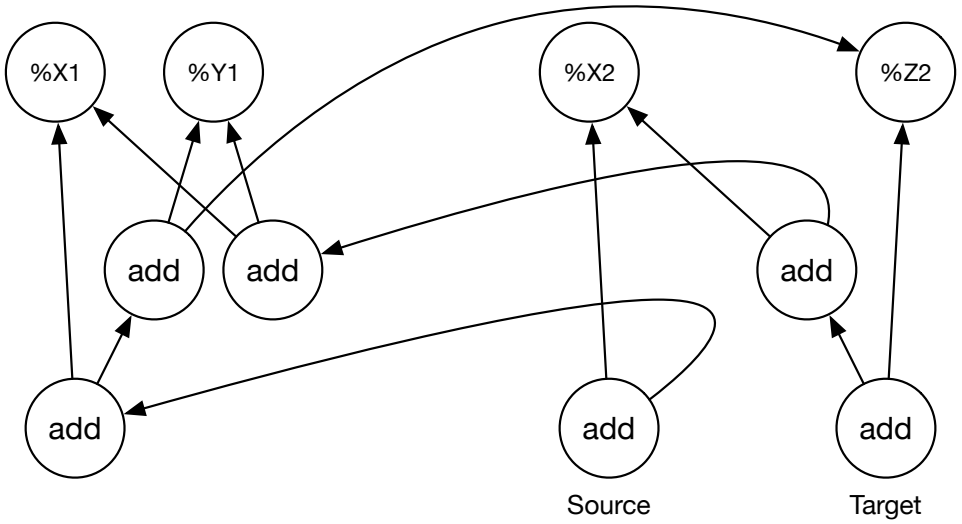
(a) A transformation that reassociates addition

```
 %b1 = add %Y1, %Z1
 %c1 = add %X1, %b1
 %c2 = add %X2, %c1
=>
 %a1 = add %X1, %Y1
 %a2 = add %X2, %a1
 %c2 = add %a2, %Z1
```

(b) Composition by matching `%c1` and `%b2`



(c) The DAGs for two copies of the transformation. Alignment begins with the root of the first target and `%b` of the second source. Dashed lines connect nodes put into the same set during alignment.



(d) The composed transformation, constructed from the second source and target. Note that the set including the first target has been replaced with the first source.

Figure 5.6: Composition of a transformation with itself, matching the root of the first instance with a non-root node of the second

Algorithm 5.4 Compose A and B, matching the root of A to b

```
1: procedure ComposeRootToNonroot(A, B, b)
2:     Sets ← AlignDAGs(A_t, b)
3:     CheckValidity(Sets)
4:     φ ← ⊤
5:     for S ∈ Sets do
6:         p, Sets ← SelectReplacement(S, Sets)
7:         φ ← φ ∧ p
8:     end for
9:     S ← Lookup(A_t, Sets)
10:    Sets ← DesignateReplacement(S, A_s, Sets)
11:    AB_s ← Graft(B_s, Sets)
12:    AB_t ← Graft(B_t, Sets)
13:    AB_p ← Graft(A_p, Sets) ∧ Graft(B_p, Sets) ∧ φ
14:    return AB
15: end procedure
```

in Figure 5.7(d) is created by rewriting the source and target of the first copy of the transformation.

5.2 Detecting Cycles

A *cycle* is a sequence of transformations that can be applied to some finite input indefinitely. The simplest such cycle is a *self-cycle* containing a single transformation. For example, if the precondition for the transformation in Figure 5.8 is satisfied for an input program, the transformation will replace the `add` with an `add nsw`. This new instruction trivially satisfies the precondition, so the transformation will be applied again, creating a new `add nsw`, and so on forever.

If the transformation's precondition had included `!hasNSW(%r)`, then the second application would be prevented and the transformation would apply only once to a given instruction.

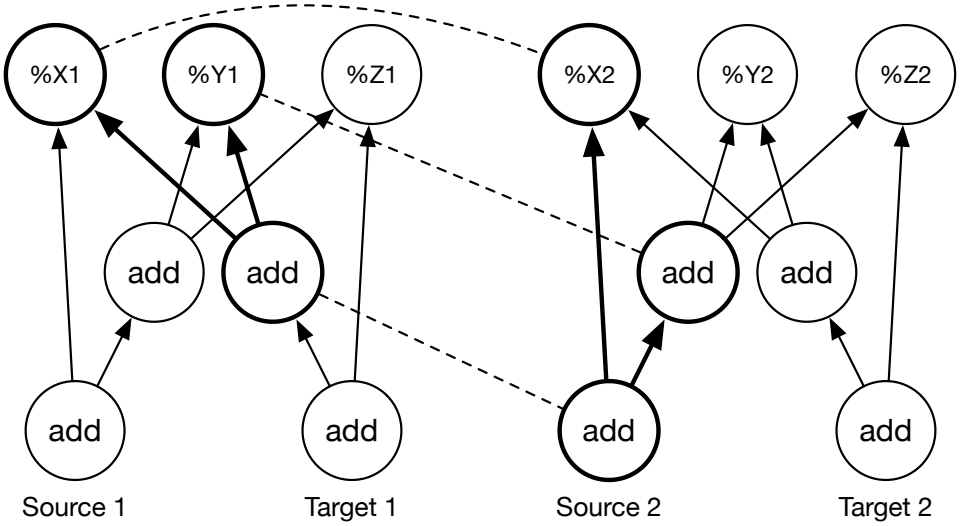
In contrast, the transformation in Figure 5.5(a) can be applied several times to a given input, but notice that the source pattern in Figure 5.5(b) is larger. That is, reassociating addition once requires two addition instructions and reassociating twice

```
  %b = add %Y, %Z
  %c = add %X, %b
=>
  %a = add %X, %Y
  %c = add %a, %Z
```

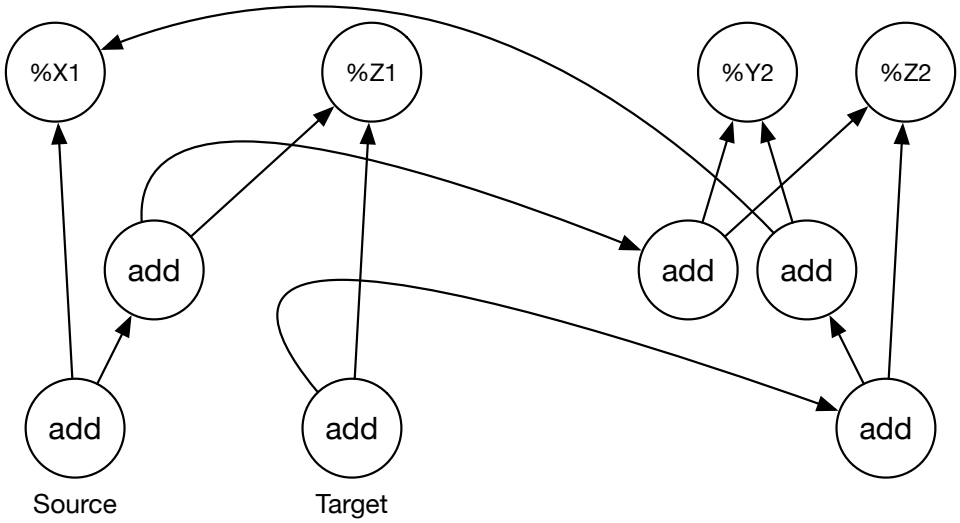
(a) A transformation that reassociates addition

```
  %b2 = add %Y2, %Z2
  %b1 = add %b2, %Z1
  %c1 = add %X1, %b1
=>
  %a2 = add %X1, %Y2
  %c2 = add %a2, %Z2
  %c1 = add %c2, %Z1
```

(b) Composition by matching `%a1` and `%c2`



(c) The DAGs for two copies of the transformation. Alignment begins with `%a` from the first target and the second target root, indicated with thick outlines. Dashed lines connect nodes put into the same set during alignment.



(d) The composed transformation, constructed from the second source and target. Note that the set including the second source has been replaced with the second target.

Figure 5.7: Composition of a transformation with itself, matching the root of the second instance with a non-root node of the first

```
Pre: WillNotOverflowSignedAdd(%a, %b)
  %r = add %a, %b
=>
  %r = add nsw %a, %b
```

Figure 5.8: A transformation that will apply indefinitely or not at all

requires three instructions. Logically, reassociating three times will require four instructions, and so forth. This ensures that the transformation can only be applied a finite number of times to a finite input program.

For a transformation to be a self-cycle, it must have these properties:

1. The precondition for its self-composition is satisfiable.

2. The source of its self-composition is no larger than the original source.

The first is easily checked by counting instructions. The second is checked by encoding the precondition of the self-composed transformation as SMT queries and checking satisfiability.

Longer sequences can be tested by creating a single composed transformation and checking whether it is a self-cycle. Composition is performed starting with the first two transformations, then composing the result with the third transformation, and continuing until a single composite transformation is produced. If it is possible for two transformations to compose in multiple ways, then all ways should be considered. A single sequence of transformations may have multiple composite transformations that represent the entire sequence. Each composite must be tested separately. If any are self-cycles, then the sequence is a cycle.

5.3 Searching for Cycles

While it is relatively simple to determine whether a given transformation sequence is a cycle, determining whether a set of transformations contains any such sequences is more daunting. In our experiments, we searched for cycles by enumerating sequences involving n transformations and testing whether they were cycles. This is not a scalable

solution—the number of sequences grows exponentially with n—but our experiments found that most longer cycles are combinations of two or more shorter cycles.

It is possible to greatly reduce the number of sequences tested for a given length. For example, a naïve enumeration of length-n sequences will include n sequences corresponding to the same cycle. For example, ABC is a cycle if and only if BCA and CAB are cycles. Thus, it is only necessary to enumerate and test one of these. Our prototype orders the set of transformations, and only generates sequences where the initial transformation is also the earliest.

A greater speed-up is obtained by memoizing the heads of the generated sequences. Sequences are generated in order, so all sequences that share a given prefix will be generated contiguously. If a given prefix does not compose, then no sequences will that prefix will compose, so it is not necessary to generate them.

However, there is a tension between these two speed-up techniques. It is theoretically possible to have a cycle involving three transformations where two of the transformations do not compose. For example, a transformation C may have a large target, and two transformations A and B may transform non-overlapping portions of that target, producing code that matches the source of C. Thus, CAB is a cycle. Even though B would not be considered to compose with A, because it does not match the target of A, it does compose with CA. This is problematic if both speed-up techniques are used. The first technique prevents generating CAB or BCA, and the second technique prevents generating ABC, because A and B do not directly compose.

This sort of situation, where two or more transformations apply to non-overlapping portions of another transformation's target, lead us to abandon a more powerful approach to finding cycles. This method worked by creating a graph whose nodes were transformations, and drawing arcs between nodes whenever one transformation composed with another.[2] Once the graph is fully populated, we can use standard techniques for finding cycles in directed graphs to generate candidate sequences: any path that starts at a node and returns to that node is potentially a cyclic transformation sequence.

[2]As it is possible for two optimizations to compose in multiple ways, this is more accurately a multigraph.

Unfortunately, this method will also miss sequences such as CAB, because there will not be an arc from A to B. While solutions can be devised for this specific problem,[3] it is unknown whether they make this method complete.

5.4 Generating Test Cases

When a cycle has been detected, it is useful to have a concrete input that demonstrates its non-termination. This aids in debugging and can be used as a test case to prevent reintroduction of the bug by future developers. The method is general, and can be used to demonstrate the behavior of any Alive-specified transformation, not just transformations produced by composing cyclic sequences.

The source of an Alive transformation is already expressed in a form similar to LLVM IR, with some additional abstractions. We obtain a valid IR fragment by replacing these abstractions with concrete values. First, we choose an arbitrary type assignment that meets the typing constraints. Second, we find values for the symbolic constants by encoding the precondition in SMT and checking its satisfiability. If it is satisfiable, we request a model, which will provide concrete values for the symbolic constants.

To turn this fragment into a valid input to LLVM, we enclose the source of the transformation in an LLVM function, with the input variables becoming parameters to the function. The value computed by the root of the source is returned from the function. Figure 5.9 shows a transformation and a corresponding concrete input.

Additional effort must be taken when the precondition includes dataflow analyses, such as `isPowerOf2`. If `%a` is an input variable, then LLVM will not be able to show `isPowerOf2(%a)`, and the transformation will not be applied. It is possible to replace inputs subject to dataflow analyses with constants that satisfy the precondition, but this is unlikely to demonstrate the transformation unless additional steps are taken. For example, when generating code for the transformation in Figure 5.8,

[3]For example: if a node C has two outgoing arcs to transformations A and B with no arcs between them, test whether CAB composes. If so, create a new node $A \oplus B$, representing the non-overlapping composition of A and B.

```
Pre: C2 == C1 & C2
  %p1 = xor %X, C1
  %r1 = and %p1, C2
=>
  %q2 = xor %X, -1
  %r1 = and %q2, C2
```

(a)

```
define i8 @foo(i8 %X) {
entry:
  %p1 = xor i8 %X, 255
  %r1 = and i8 %p1, 0
  ret i8 %r1
}
```

(b)

Figure 5.9: A transformation (a) and a concrete input program (b) that will cause non-termination. The process instantiated types as 8-bit integers and chose the values `255` and `0` for `C1` and `C2`, respectively.

we could replace `%a` and `%b` with `0` and `0`, respectively. The resulting precondition, `WillNotOverflowSignedAdd(0,0)` will be satisfied, but the instruction `add i8 0, 0` will be replaced with `0` by InstCombine's constant folding and the transformation will not apply. Thus, this method requires disabling constant folding in order to demonstrate non-termination.

A more general solution is to generate additional code that ensures that the dataflow predicates are satisfied. For example, `isPowerOf2(%x)` is satisfied if `%x = shl 1, %y`, for some new variable `%y`. Similarly, `%a` and `%b` will not overflow if their upper bits are zero, due to an `and` or from being the result of `zext`. This method becomes more difficult if the same input variable is the subject of multiple analyses, but this does not commonly occur.

5.4.1 Shadowing of transformations

The technique for finding cycles will return sequences of transformations that can be applied indefinitely to some input, but it is possible that this exact sequence of transformations will never be applied to any input. Because the InstCombine tests potential transformations in an specific order and selects the first one that matches, many transformations will only be applied to a subset of possible inputs. If two transformations can apply to a given program, only the first will ever be performed. We say that the first has *shadowed* the second for that program.

For a cycle to cause compiler non-termination, each transformation in the cycle must be applied to the program resulting from the previous transformation. If any

```
Pre: C < 0 && isPowerOf2(abs(C))
  %p = sub %Y, %X
  %r = mul %p, C
=>
  %q = sub %X, %Y
  %r = mul %q, abs(C)
```

(a) A transformation

```
Pre: C < 0 && isPowerOf2(abs(C)) \
     && abs(C) < 0 \
     && isPowerOf2(abs(abs(C)))
  %p = sub %Y, %X
  %r = mul %p, C
=>
  %q = sub %X, %Y
  %r = mul %q, abs(abs(C))
```

(b) The self-composition of (a)

```
define i4 @foo(i4 %X, i4 %Y) {
entry:
  %p = sub i4 %Y, %X
  %r = mul i4 %p, 8
  ret i4 %r
}
```

(c) Generated test case

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

(d) An optimization shadowing (a)

Figure 5.10: An optimization that can be applied to an input indefinitely. Note that `abs(C) < 0` is satisfied when `C` is the minimum signed integer, so the precondition of (b) can be satisfied. However, the minimum signed integer is also a power of two, so any input that satisfies (b) also satisfies (d). Thus, if (d) is attempted before (a), the cycle is prevented.

of these transformations are shadowed, then a transformation outside the cycle may be performed, resulting in a program that does not trigger the cycle. This means it is possible that a cycle in a set of transformations will not cause compiler non-termination in an actual implementation. Figure 5.10 shows a transformation that is a self-cycle and another transformation that shadows it, preventing the cycle from occurring.

As described, both non-termination testing and concrete input generation assume that no shadowing occurs. This is not generally the case. This means that the cycle detector may incorrectly report some sequences as being cycles, and the concrete input generator may produce inputs that do not cause non-termination. The former is not necessarily a problem, as the concrete inputs can be used to determine whether non-termination occurs. The risk of the second can be reduced by producing multiple concrete inputs choosing different values.

A more complete solution is to strengthen the preconditions to reflect shadowing. Each transformation has an implicit precondition stating that no earlier transformation

applied. If this condition can be made explicit, then the existing techniques will work without modification. For the example in Figure 5.10, the shadowing can be indicated by adding the predicate `!isPowerOf2(C)` to Figure 5.10(a). This will give the self-composite in Figure 5.10(b) an unsatisfiable precondition, so it will not be reported as a cycle.

Not all shadow preconditions are currently expressible in Alive. For example, a transformation may implicitly assume that an input variable `%x` is not an **add** instruction. This can be addressed by adding additional predicates, but further research is needed to know whether this is sufficient to express all implicit shadow preconditions.

5.5 Evaluation

To assess the efficacy of this technique, we extended the Alive toolkit with a prototype termination checker and searched for cycles among a set of transformations derived from the LLVM's InstCombine pass. We had observed non-termination when testing a peephole optimization pass created using Alive's code generator, so we expected to find several cycles within that set of transformations. For any cycles found, we created concrete programs that could be used with LLVM to determine whether this non-termination occurred in a production compiler.

5.5.1 Methodology

The termination checker was built on the Alive toolkit, adding roughly 1800 lines of Python code. Given a suite of transformations, it generates sequences of a specified length containing no duplicate transformations. To avoid testing multiple sequences representing the same cycle, only sequences where the first transformation occurs earliest in the suite are generated. That is, the prototype generates sequences ABC and ACB, but not BAC, BCA, CAB, or CBA, all of which are redundant. Sequence generation and composition were interleaved, so that a single composition AB would be reused for all sequences beginning with AB. This allows the detector to skip all

sequences beginning with a common prefix if that common prefix does not viably compose, as those sequences could not possibly be cycles.

To take advantage of multiprocessing, the search process was broken into a client-server architecture when searching for n-cycles for n larger than three. A manager process breaks the set of all sequences of n transformations into chunks sharing a common prefix and makes these prefixes available as a work queue. Client processes obtain prefixes from the queue, check all sequences beginning with that prefix for potential non-termination, and report any discoveries to the manager, along with statistics such as the number of sequences considered. Once the work queue is emptied, the clients terminate.[4] The number of cycles grows exponentially with the length of cycles, so even with multi-processing it was only practical to test all sequences for small n. For larger n, the manager selects prefixes randomly, rather than sequentially, and halts after one million composable sequences have been examined.

Searches were performed using a suite of 416 transformations derived from InstCombine, including transformations using integer arithmetic but not floating-point arithmetic or memory operations. SMT queries were resolved using a development version of Z3 available at the time, as it had better support for quantifiers than the then-current official release. Experiments were performed on four-core, 64-bit Intel Haswell machines with 16 GiB of RAM.

5.5.2 Experimental results

The prototype successfully found 184 cycles among the 416 transformations in the Alive suite, including those that had previously been observed to cause non-termination in the Alive-generated peephole optimizer. Table 5.1 gives statistics about the sequences considered during the search, including the number of sequences considered (including all sequences that were discarded because a prefix was not composable), the number of distinct composite transformations produced, the number of those that could apply to

[4]To mitigate a space leak in the version of Z3 used for this testing, clients also terminated after testing a certain number of sequences, at which point the manager would start a new client process.

n	Transformation Sequences	Complete Compositions	Self-compositions	Non-increasing	Cycles Found
1	416	416	296	25	23
2	86 320	7 001	4 292	31	27
3	23 824 320	182 678	96 989	49	35
4	7 379 583 120	5 524 634	2 694 291	152	99
5*	13 119 902 905	1 000 000	463 017	2	0
6*	97 613 680 549	1 000 000	394 794	0	0
7*	474 163 216 578	1 000 000	395 638	0	0
			Total Number of Cycles		**184**

Table 5.1: Statistics for the experiment run with the Alive suite. When generating a sequence of n transformations, the columns give (2) the number of sequences explored, (3) the number of transformations found by composing sequences, (4) the number of such transformations that can be applied to their own output, (5) the number that do not increase the size of the source pattern under self-composition, and (6) the number of cycles.
* indicates that sequences were randomly sampled, rather than exhaustively searched.

their own output, the number that did not require a larger input to do so, and the number that then had a satisfiable precondition. As noted earlier, the number of sequences grows rapidly with the sequence length, but the ratios between each column reduce. Exhaustive searches were only performed for sequences of up to four transformations. Larger sequence lengths were randomly sampled until a million composable sequences were examined.

Characterization of cycles

Figures 5.11 to 5.13 show some of the transformations involved in cycles found by the prototype. Surprisingly, the 184 cycles found involve only 38 distinct transformations. Many of the larger cycles involve multiple transformations that participate in smaller cycles. For example, the four transformations in Figure 5.13 form two distinct 4-cycles, and two additionally form 1-cycles. Similarly, the transformation in Figure 5.8 can form a 2-cycle with several other 1-cycles involving an **and** instruction. Of the 184 cycles, only 32 did not incorporate a smaller cycle. This suggests that exploring small sequence lengths is sufficient to find the majority of non-termination bugs.

The cycles are enabled by weak preconditions, either because the precondition was deliberately weakened during translation to Alive or because the precondition involved

```
Name: AndOrXor 2
  %op = or %X, C1
  %r  = and %op, C2
=>
  %o  = or %X, (C1 & C2)
  %r  = and %o, C2
```

(a)

```
Name: AndOrXor 5
Pre: C2 & (-1 u>> C1) != -1 u>> C1
  %op = lshr %X, C1
  %r  = and %op, C2
=>
  %r  = and %op, C2 & (-1 u>> C1)
```

(b)

```
Name: AndOrXor 13
  %op0 = or %A, C1
  %r   = or %op0, %op1
=>
  %i   = or %A, %op1
  %r   = or %i, C1
```

(c)

```
Name: AndOrXor 8
Pre: MaskedValueIsZero(%A, \
    -1 u>> countLeadingZeros(C))
  %lhs = sub %A, %B
  %r   = and %lhs, C
=>
  %neg = sub 0, %B
  %r   = and %neg, C
```

(d)

```
Name: Select 1
  %c = icmp eq %X, C
  %r = select i1 %c, %X, %Y
=>
  %r = select i1 %c, C, %Y
```

(e)

```
Name: Select 2
  %c = icmp ne %X, C
  %r = select i1 %c, %Y, %X
=>
  %r = select i1 %c, %Y, C
```

(f)

Figure 5.11: A sampling of the optimizations that form 1-cycles. The transformation in Figure 5.8 also forms a 1-cycle.

```
Name: AndOrXor 9
  %op0 = xor %nOp0, -1
  %op1 = xor %nOp1, -1
  %r = and %op0, %op1
=>
  %or = or %nOp0, %nOp1
  %r = xor %or, -1
```

```
Name: AndOrXor 12
  %na = xor %A, -1
  %nb = xor %B, -1
  %r = or %na, %nb
=>
  %a = and %A, %B
  %r = xor %a, -1
```

```
Name: AndOrXor 15
  %op0 = or %x, %y
  %r = xor %op0, -1
=>
  %nx = xor %x, -1
  %ny = xor %y, -1
  %r = and %nx, %ny
```

(a)

```
Name: AndOrXor 14
  %op0 = and %x, %y
  %r = xor %op0, -1
=>
  %nx = xor %x, -1
  %ny = xor %y, -1
  %r = or %nx, %ny
```

(b)

Figure 5.12: A sampling of the optimizations that form 2-cycles. The transformations in Figures 5.3(a) and 5.3(b) also form a 2-cycle.

predicates not yet added to Alive and unnecessary for correctness, such as **isConstant** or **hasNSW**. The 1-cycle shown in Figure 5.11(a) can be broken by adding the precondition `C1 != C1 & C2`. (This will also break several larger cycles that include this transformation.) Similarly, adding the precondition `!isConstant(%A)` will break the cycle shown in Figure 5.11(d).

Demonstrating non-termination

A concrete LLVM IR input program was generated for each detected cycle and given to LLVM's InstCombine pass and to an Alive-generated peephole optimizer. To avoid interference from other optimization passes, only the peephole optimizer was used. The Alive-generated optimizer did not terminate for 179 of the 184 cycle-triggering input programs. In the remaining five cases, optimization terminated due to interference with other transformations, as described in Section 5.4.1.

None of the concrete inputs caused non-termination in InstCombine. This is both because InstCombine includes more transformations, and thus more opportunities for interference, and because the transformations involved in cycles often had stronger preconditions in InstCombine than the corresponding transformations in the Alive suite.

```
Name: AndOrXor 2
  %op = or %X, C1
  %r  = and %op, C2
=>
  %o  = or %X, (C1 & C2)
  %r  = and %o, C2
```

(a)

```
Name: AndOrXor 3
Pre: C1 & C2 == C1
  %op = or %X, C1
  %r  = and %op, C2
=>
  %a  = and %X, C2^(C1&C2)
  %r  = or %a, C1
```

(b)

```
Name: AndOrXor 10
Pre: C & C1 != 0
  %op0 = and %x, C1
  %r   = or %op0, C
=>
  %or  = or %x, C
  %r   = and %or, (C | C1)
```

(c)

```
Name: AndOrXor 13
  %op0 = or %A, C1
  %r   = or %op0, %op1
=>
  %i   = or %A, %op1
  %r   = or %i, C1
```

(d)

```
Pre: C12 & C11 != 0 && \
    C12 & (C12 | C11) == C12
  %op = or %X, C1
  %r  = and %op, C11
  %r1 = or %r, C12
  %r2 = or %r1, %op1
=>
  %o  = or %X, (C1 & C11)
  %a  = and %o, (C12 | C11) ^ \
    (C12 & (C12 | C11))
  %i  = or %a, %op1
  %r2 = or %i, C12
```

(e)

```
Pre: C111 & C2 & C1 != 0 && \
    C111 & C2 & ((C111&C2)|C1) \
      == C111 & C2
  %op0  = and %x, C1
  %op01 = or %op0, C11
  %op   = or %op01, C111
  %r    = and %op, C2
=>
  %a   = and %x, ((C111&C2)|C1) ^ \
    (C111 & C2 & ((C111&C2)|C1))
  %r1  = or %a, (C111 & C2)
  %r11 = or %r1, C11
  %r   = and %r11, C2
```

(f)

Figure 5.13: Four transformations participating in two 4-cycles. (e) composes the sequence (a)-(c)-(b)-(d); (f) composes the sequence (a)-(d)-(c)-(b). Note that (a) and (d) form 1-cycles.

Specifically, during translation to Alive, the preconditions for several transformations were weakened by removing clauses that were not needed for correctness. These clauses turned out to be necessary for preventing non-termination. Once again, this indicates that a weaker precondition is not always preferable.

5.6 Summary

Optimizations can fail in two ways: they may produce incorrect results by changing the meaning of a program, or they may fail to produce any results due to non-termination or other errors. A collection of individually correct transformations is not vulnerable to the first kind of failure, but may exhibit the second. While termination checking is undecidable in general, the limited domain of Alive transformations makes it feasible to determine whether a particular sequence of transformations can be applied indefinitely to some input. More generally, we see that using a declarative language for specifying peephole optimizations enables multiple analyses in addition to verifying correctness.

Chapter 6

Related Work

Work related to Alive can be divided into three categories: (1) software verification and compiler correctness, (2) code inference, especially precondition inference, and (3) and termination checking.

6.1 Compiler Correctness

Most approaches for improving compiler correctness can be classified as testing tools, formal reasoning frameworks, and domain specific languages (DSLs). In addition to Alive, DSLs for compiler optimizations include languages based on graph rewriting [9, 79, 101], regular expressions [61], computation tree logic (CTL) [63], type systems [104], and rewrite rules [46, 66, 67, 125]. In particular, Alive is similar to high-level rewrite patterns [62, 78], but it differs from earlier approaches by including strong reasoning about undefined behavior in order to verify modern optimizations that exploit it.

An alternative to verifying hand-written optimizations is automatic generation of peephole optimizations [26] or superoptimization [11, 56, 81, 98, 109, 119], which finds equivalent programs that use the fewest instructions.

Optgen [18] automatically generates peephole optimizations over an IR that are verified using an SMT solver. In contrast to Alive, Optgen handles only integer operations, does not reason about undefined behavior, and does not abstract over types.

Tools for random testing [10, 64, 72, 89, 127] have successfully found many bugs in compilers, but can never provide assurance that all bugs have been found—as demonstrated by the fact that Alive found incorrect transformations in InstCombine that had been missed by previous random testing projects.

Translation validation [51, 93, 99, 102, 105, 111, 118, 123, 128, 132] prevents a compiler from producing incorrect code by comparing the input and output to a particular invocation of the compiler and showing that they are equivalent. Often, showing this equivalence is easier than verifying the correctness of the compiler itself, but it requires a proof to be found for each compilation. This degree of overhead is unlikely to be accepted in a mainstream compiler.

CompCert [68–70, 90, 120] is a C compiler written using the proof assistant Coq [23] that uses a combination of verified code and translation validation to ensure correctness. Additional work has formalized optimizations involving weak memory models [124] and bit-precise floating-point operations [14]. Vellvm [130, 131] also uses Coq, but models the semantics of LLVM IR. Alive treats `undef` values similarly to Vellvm, but can handle poison values and uses an SMT solver for correctness checking.

Extensive prior work has been done to check or improve the precision of floating-point computation [13, 16, 25, 27, 35, 42, 43, 53, 59, 97, 103, 115], which is related to but distinct from verification of compiler optimizations. In the absence of "fast math" attributes, Alive restricts itself to considering bit-precise computations [42, 50, 80, 88].

LifeJacket [95] is another project that extended Alive with floating-point operations, developed concurrently with Alive-FP [87]. LifeJacket uses a single encoding of floating-point operations, similar the `undef`-like interpretation of undefined results. Its encoding of **nsz** loosens the value equality correctness check to allow a change in the sign of a zero result, which is less flexible than the encoding used by Alive.

6.2 Precondition Inference

Considerable work has been done on inferring preconditions, postconditions, and invariants for general-purpose programs [5, 7, 8, 12, 24, 31, 34, 36–38, 96, 106, 110, 112], including data-driven approaches [36–38, 96, 106].

Alive-Infer is inspired by PIE [96], which generates preconditions in a general setting. Precondition generation is broken into two steps: feature learning, which finds

predicates, and Boolean formula learning, which uses the predicates to create a precondition. Alive-Infer works in the context of LLVM and Alive. Its language of predicates is expressed using bit-vector arithmetic instead of integer arithmetic, and includes terms that are partially defined. Alive-Infer allows developers to choose between full preconditions that may be complex and simpler partial preconditions that may reject some positive examples. To support this, Alive-Infer introduces a weighted partial Boolean formula learner. Alive-Infer learns predicates that are type parametric and generates data for multiple type assignments in order to capture type-varying behavior.

Other work has specifically addressed precondition generation for compiler optimizations [18, 77, 108]. PSyCO [77] synthesizes read-write preconditions, but requires a finite set of predicates determined in advance and does not address the features of bit-vector arithmetic or undefined behavior. Optgen [18] automatically generates peephole optimizations and checks their correctness. These may include preconditions, but the language of precondition is limited to requiring that some computation evaluate to zero. Preconditions are found using enumeration.

An alternative to finding predicates through enumeration is logical abduction [31, 39], which derives preconditions using techniques such as quantifier elimination. Methods for quantifier elimination in bit-vector logic [55] are limited to linear arithmetic, which is insufficient to find preconditions for many Alive transformations.

Many previous data-driven approaches require a set of predicates to be fixed in advance [38, 106, 112]. Counter-example–guided refinement [21, 110] is similar to Alive-Infer, separating sets of mixed positive and negative examples by finding counter-examples, but also fixes the set of predicates in advance. ICE and ICE-DT [36, 37] introduce implication examples alongside positive and negative examples, using all three to synthesize invariants. Specific techniques may involve template-based synthesis or a decision tree learning algorithm generating invariants using a fixed set of attributes. In contrast, PIE and Alive-Infer learn predicates on-demand.

Alive-Infer can also be seen as a variant of the symbolic, stochastic, and enumerative search strategies used for program synthesis [6, 44, 60, 114, 122] and superoptimization.

6.3 Termination Checking

Few of the existing approaches for compiler verification address non-termination A compiler written with end-to-end verification in a proof assistant will be guaranteed to terminate by the proof assistant. DSLs typically address the correctness of individual optimizations and do not address non-termination that can arise when a set of transformations is run until it reaches a steady state. Conversely, translation validation needs the output of the compiler in order to test for correctness, and provides no assurances about the process of compilation itself.

In a broader context, there is extensive prior work on termination checking for imperative programs, term-rewriting systems, system specifications, and system code [19, 20, 22, 45, 58, 71, 116, 116]. These methods discover invariants statically or dynamically such as ranking functions and use them to show termination. Finding such invariants for Alive-generated peephole optimizers is challenging, as it would need to consider the LLVM infrastructure used by InstCombine.

Research for showing termination in term-rewriting systems [30, 40, 117, 126] is relevant to Alive, but cycle detection using composition is able to take advantage of the structure and domain knowledge of Alive.

Non-termination can also be seen as an extreme case of poor performance. If Alive is configured to generate transformations that cease applying after some limit is reached, the time spent in the cycle can be analyzed by tools that detect the cause of poor performance [29, 48, 94]. However, these are dynamic analyses, and cannot be applied without a concrete input that triggers the cycle. In contrast, Alive termination checking is a static analysis that can generate concrete inputs that trigger cycles after they have been detected.

Chapter 7

Conclusion

The goal of the Alive project is to encourage software developers, specifically compiler developers, to demonstrate the correctness of their code by simplifying the task of creating a rigorous proof. A peephole transformation written in Alive can be automatically checked for correctness and translated to a C++ implementation, as described in Chapter 3. Alive transformations are often shorter than hand-written transformations in C++, and their effect can be easily determined. If the Alive toolkit determines that a transformation is incorrect, it will provide an example showing how the transformation changes the behavior of a program. The Alive-Infer toolkit described in Chapter 4 can then be used to find a precondition that makes the transformation correct, if one exists.

Alive is designed for automated verification. It supports a subset of instructions in the LLVM IR and can describe a specific class of transformations, but within these parameters it is able to check correctness without user intervention. The actual correctness conditions can be efficiently checked by SMT solvers, often in less than a second.

In addition to showing the correctness of individual transformations, Alive enables higher-level analyses of multiple transformations. A peephole optimizer that repeatedly applies transformations until no further transformations can be applied is vulnerable to non-termination bugs, where no steady state is ever reached. If the transformations making up the optimizer are specified in Alive, the method described in Chapter 5 can be used to detect sequences of transformations that may lead to non-termination. The declarative nature of Alive means it is always possible to automatically determine the effect of applying a transformation to a program, which is not guaranteed for transformations written in a general-purpose programming language.

In short, Alive is a practical system for creating correct peephole transformations.

It is fast, easy to use, and rigorous. Alive has been adopted by LLVM developers for checking the correctness of proposed transformations and as a way to explore the semantics of LLVM IR.

7.1 Technical Contributions

The Alive language, described in Chapter 3, selects a subset of LLVM IR for which the correctness of peephole transformations can be automatically checked. To perform these checks, we introduce an method for encoding the semantics of several LLVM instructions and expressions into SMT. This includes encodings of LLVM's three forms of undefined behavior as well as a safety condition that ensures the transformation itself can be applied without causing undefined behavior in the compiler and verified without causing undefined behavior in the SMT solver. Alive uses side conditions to ensure that its encoding of imprecise analyses represent their approximate nature.

The semantics for floating-point operations in LLVM are vague, especially those relating to its "fast math" flags. Alive encodes transformations assuming IEEE 754 semantics and introduces a formal, instruction-level encoding of the `nnan`, `ninf`, and `nsz` attributes.

Alive-Infer, described in Chapter 4, introduces a data-driven method for inferring preconditions sufficient to make a transformation correct. This method expands on the Precondition Inference Engine by generating data for abstract programs (the source pattern of Alive transformations) that cannot themselves be executed. Alive-Infer learns predicates by enumerating predicates that are well-typed and type parametric, and can reason about predicates that are partially defined (*i. e.*, have non-trivial safety conditions). Alive-Infer introduces a weighted partial Boolean formula learner, which finds formulae that reject all negative examples and maximize the number of accepted positive examples, while maintaining an upper bound on formula complexity.

Chapter 5 presents a method for composing two transformations to create a single transformation that has the effect of applying both transformations to an input program in sequence. Because Alive transformations are declarative, this process can be

performed by aligning directed, acyclic graphs. Using composition, it is possible to check whether a sequence of transformations can be applied indefinitely to some input program by comparing the composite transformations that apply the entire sequence once and twice. If performing the sequence twice does not increase the size of the source pattern compared to performing it once and its precondition can be satisfied, then the transformation sequence can be performed indefinitely.

7.2 Future Work

The Alive project has successfully created a practical system for verifying peephole optimizations that has been adopted by mainstream compiler developers, but there are many avenues left to explore.

Alive itself can be extended in several directions beyond those discussed in Section 3.5. Several InstCombine transformations target LLVM intrinsics, such as `fabs` (floating-point absolute value). Syntax for LLVM intrinsics could be easily extended in the same way as constant and predicate functions, allowing for easily adding new intrinsics to Alive without modifying the parser.

In our experiments, we found that transformations targeting `getelementpointer` instructions are applied more frequently than those targeting any other instruction. Several transformations abstract over lists of arguments, which requires additional syntax to represent in Alive. It is worth investigating what would be needed to express these transformations in Alive, and whether the gain in expressive power justifies the additional syntax and semantics.

Similarly, several InstCombine transformations are parametric over classes of instructions (*e. g.*, several transformations apply to both signed and unsigned division). It would be possible to allow abstraction over binary operators, similar to the relation variables discussed in Section 3.5.1, with additional predicates to restrict the range of possible instructions.

The "fast math" flags in LLVM can be further explored. Beyond requiring bit-precise transformations or allowing a certain degree of nondeterminism, the behavior of

the flags can also be characterized in terms of lost precision. Floating-point arithmetic is inherently imprecise, so alternative encodings could be developed that compare the error bounds of the source and target of a transformation. There are many ways to define these error bounds (average vs maximum error, relative vs absolute error, etc.), and not all can be encoded equally efficiently. Such a project would need to consult not only with numerics experts, but with the users of the fast math flags in order to determine their needs.

More broadly, the need to separately check correctness for each possible type assignment is unsatisfying, both because it requires dozens or hundreds of correctness checks and because even those checks do not actually show correctness for all $2^{23} - 1$ possible integer types. Some means of showing that an SMT problem is unsatisfiable for all bit widths larger than a threshold would remove this small threat to the validity of Alive verification, as well as being useful for other applications of SMT bit vectors.

The inference algorithm used by Alive-Infer is effective, but it can struggle to find predicates to separate some samples. The exponential growth in the number of predicates as the predicate size increases makes it impractical to expect enumeration of predicates above a certain size. When Alive-Infer takes a very long time to find a precondition, it is usually because it cannot learn a large predicate without first examining all smaller predicates. Presently, the enumerator is guided only by the names of symbolic constants and type variables defined by a transformation, but it may be possible to find additional information in the transformation and use it to bias the enumerator to produce predicates more likely to be relevant sooner. For example, most integer constants will be intended as signed or unsigned values, and expressions in the precondition that respect this intention are more likely to be useful.

Finally, detection of compiler non-termination bugs is only one possible global analysis that could be performed on a collection of Alive transformations. Many transformations have implicit assumptions based on their priority in InstCombine, which may lead to preconditions that appear insufficient or partial when taken individually or imply cycles that cannot occur due to shadowing. Methods similar to composition could be developed to discover these implicit assumptions, avoiding these incorrect analyses

and also enabling discovery of transformations that will never be applied. Similarly, knowing that two transformations overlap can be help to determine the best ordering of transformations, as transformations that produce better code or are easier to perform should have higher priority.

When generating a peephole optimizer, the code generator organizes transformations by the opcode of their target instruction but otherwise generates code for each transformation separately. Often, several transformations have source patterns with common sub-patterns. Rather than checking for the same pattern multiple times, a more efficient implementation would attempt to match the pattern once and share the result among the transformations. Again, DAG alignment can be extended to find transformations that would benefit from sharing sub-pattern matches and the peephole optimizer can be constructed to minimize duplicated effort.

7.3 Summary

The goal of the Alive project is to encourage the use of formal methods among compiler developers by creating tools that are simple to use while providing a strong assurance of correctness. Alive provides a way to specify peephole transformations that can be automatically checked for correctness and produce implementations that are correct by construction. Building on Alive, we also provide a method for debugging incorrect Alive transformations by inferring preconditions that make a transformation correct, and another for detecting sequences of transformations that may cause compiler non-termination. Although Alive cannot be used to show the end-to-end correctness of a compiler, it and its associated tools have successfully improved the correctness of LLVM. Alive has been used to find previously unknown bugs in LLVM and, more importantly, has been adopted by LLVM developers as a means of screening proposed additions to LLVM's peephole optimizer. This success is not the end of the Alive project: Alive can form the basis of additional analyses and the ideas behind Alive can be adapted to other compilers and intermediate languages.

Bibliography

[1] Alive optimization verifier. `https://rise4fun.com/Alive`. Retrieved 2017-10-11.

[2] LLVM language reference manual. `http://llvm.org/docs/LangRef.html`. Retrieved 2017-10-14.

[3] IEEE standard for floating-point arithmetic. IEEE 754-2008, IEEE Computer Society, Aug. 2008.

[4] W. Ackermann. *Solvable Cases of the Decision Problem*. North-Holland Publishing Company, 1954.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] A. Balestrat. CCG: A random C code generator. `https://github.com/Merkil/ccg/`. Retrieved 2016-02-12.

[11] 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.

[12] 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.

[13] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 549–560, New York, NY, USA, 2013. ACM.

[14] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In *Proceedings of the 21st IEEE Symposium on Computer Arithmetic*, ARITH, pages 107–115. IEEE, Apr. 2013.

[15] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and software technology*, 39(9):617–625, 1997.

[16] M. Brain, V. DSilva, A. Griggio, L. Haller, and D. Kroening. Interpolation-based verification of floating-point programs with abstract CDCL. In *Proceedings of the 20th International Symposium on Static Analysis*, SAS, pages 412–432. Springer, June 2013.

[17] M. Brain, C. Tinelli, P. Rümmer, and T. Wahl. An automatable formal semantics for IEEE-754 floating-point arithmetic. In *Proceedings of the 22nd IEEE Symposium on Computer Arithmetic*, ARITH, pages 160–167. IEEE, June 2015.

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

[19] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 161–169, 2009.

[20] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with Jolt. In *Proceedings of the 25th European conference on Object-oriented programming (ECOOP)*, pages 609–633, 2011.

[21] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *CAV*, 2000.

[22] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426, 2006.

[23] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2013.

[24] 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.

[25] E. Darulova and V. Kuncak. Sound compilation of reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 235–248, New York, NY, USA, 2014. ACM.

[26] J. W. Davidson and C. W. Fraser. Automatic generation of peephole transformations. In *Proc. of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 111–115, New York, NY, USA, 1984.

[27] F. de Dinechin, C. Q. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC, pages 1318–1322. ACM, 2006.

[28] 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.

[29] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 516–527, 2011.

[30] N. Dershowitz. Termination of rewriting. *Journal of symbolic computation*, 3(1):69–115, 1987.

[31] 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.

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

[33] E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, pages 255–264, New York, NY, USA, 2008. ACM.

[34] 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.

[35] Z. Fu, Z. Bai, and Z. Su. Automated backward error analysis for numerical code. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 639–654, New York, NY, USA, 2015. ACM.

[36] 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.

[37] 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.

[38] 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.

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

[40] J. Giesl, P. Schneider-kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR)*, pages 281–286, 2006.

[41] D. Gohman. The nsw story. `http://lists.llvm.org/pipermail/llvm-dev/2011-November/045735.html`, Nov. 2011. Retrieved 2017-11-28.

[42] E. Goubault. Static analyses of the precision of floating-point operations. In

Proceedings of the 8th International Symposium on Static Analysis, SAS, pages 234–259. Springer, 2001.

[43] E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: Principles and experiments. In *Revised Selected Papers from the 12th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS, pages 3–20. Springer, 2007.

[44] 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.

[45] A. Gupta, T. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 147–158, 2008.

[46] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, Feb. 2005.

[47] M. Haidl. [llvm-dev] How to add optimizations to InstCombine correctly? `http://lists.llvm.org/pipermail/llvm-dev/2017-September/117419.html`, Sept. 2017. Retrieved 2017-10-24.

[48] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 145–155, 2012.

[49] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[50] J. Harrison. Floating point verification in HOL. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 186–199. Springer, Sept. 1995.

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

[52] T. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, Jan. 2003.

[53] F. Ivančić, M. K. Ganai, S. Sankaranarayanan, and A. Gupta. Numerical stability analysis of floating-point computations using software model checking. In *Proceedings of the 8th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE, pages 49–58. IEEE, July 2010.

[54] Y. Jiang. [Patch]InstCombine pattern for ICMP. `http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20140818/231300.html`, 2014. Retrieved 2017-09-14.

[55] 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.

[56] 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.

[57] 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.

[58] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Detecting liveness bugs in systems code. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NDSI)*, pages 243–256, 2007.

[59] A. B. Kinsman and N. Nicolici. Finite precision bit-width allocation using SAT-modulo theory. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1106–1111, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[60] 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.

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

[62] 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.

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

[64] 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.

[65] 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 *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 633–647, New York, NY, USA, June 2017. ACM.

[66] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI, pages 220–231, 2003.

[67] 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.

[68] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages*, 2006.

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

[70] X. Leroy. A formally verified compiler back-end. In *Journal of Automated Reasoning*, 2009.

[71] P. Li and J. Regehr. T-check: Bug finding for sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 174–185, 2010.

[72] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 65–76, 2015.

[73] C. Lindig. Random testing of c calling conventions. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 3–12. ACM, 2005.

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

[75] N. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Alive: Automatic LLVM InstCombine Verifier. `http://github.com/nunoplopes/alive`. Retrieved 2016-02-12.

[76] 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.

[77] 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.

[78] N. P. Lopes and J. Monteiro. Automatic equivalence checking of uf+ia programs. In *Proceedings of the 20th International Symposium on Model Checking Software*, SPIN, pages 282–300, July 2013.

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

[80] M. Martel. Semantics-based transformation of arithmetic expressions. In *Proceedings of the 14th International Symposium on Static Analysis*, SAS, pages 298–314. Springer, 2007.

[81] 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.

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

[83] D. Menendez. LLVM's shifty semantics. `https://compilersatrutgers.wordpress.com/2017/05/24/llvms-shifty-semantics/`, May 2017. Retrieved 2017-10-11.

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

[85] 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.

[86] D. Menendez and S. Nagarakatte. Precondition inference for peephole optimizations in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 49–63, June 2017.

[87] 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.

[88] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):12:1–12:41, May 2008.

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

[90] 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.

[91] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53, 2012.

[92] E. Nagai, A. Hashimoto, and N. Ishiura. Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013)*, pages 88–93, 2013.

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

[94] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. CARAMEL: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 902–912, 2015.

[95] 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.

[96] 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.

[97] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 1–11. ACM, June 2015.

[98] 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.

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

[100] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[101] N. Ramsey, J. Dias, and S. P. Jones. Hoopl: A modular, reusable library for

dataflow analysis and transformation. In *Proc. of the 3rd ACM Symposium on Haskell*, 2010.

[102] M. Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, Massachusetts Institute of Technology, Mar. 1999.

[103] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 27:1–27:12, New York, NY, USA, 2013. ACM.

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

[105] H. Samet. Proving the correctness of heuristically optimized code. *Communications of the ACM*, 21(7):570–582, July 1978.

[106] 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.

[107] R. L. Sauder. A general test data generator for COBOL. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 317–323. ACM, 1962.

[108] 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.

[109] 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.

[110] 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.

[111] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[112] 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.

[113] F. Sheridan. Practical testing of a C99 compiler using output comparison. *Software: Practice and Experience*, 37(14):1475–1488, Nov. 2007.

[114] 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.

[115] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In *Proceedings of the 20th International Symposium on Formal Methods*, FM, pages 532–550. Springer, June 2015.

[116] F. Spoto, F. Mesnard, and É. Payet. A termination analyzer for Java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(3):8:1–8:70, Mar. 2010.

[117] J. Steinbach. Simplification orderings: History of results. *Fundamenta Informaticae*, 24(1–2):47–87, Apr. 1995.

[118] M. Stepp, R. Tate, and S. Lerner. Equality-Based translation validator for LLVM.

In *CAV '11: Proceedings of the 23rd International Conference on Computer Aided Verification*, 2011.

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

[120] 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.

[121] C. Topper. [llvm-dev] where did alive go? `http://lists.llvm.org/pipermail/llvm-dev/2017-September/117601.html`, Sept. 2017. Retrieved 2017-10-11.

[122] 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.

[123] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for llvm. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2011.

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

[125] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):1053–1084, Nov. 1997.

[126] H. Xi. Towards automated termination proofs through "freezing". In *Rewriting Techniques and Applications*, pages 271–285. Springer, 1998.

[127] 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.

[128] A. Zaks and A. Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *FM '08: Proceedings of the 15th international symposium on Formal Methods*, pages 35–51, Berlin, Heidelberg, 2008. Springer-Verlag.

[129] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang. Automated test program generation for an industrial optimizing compiler. In *Automation of Software Test, 2009. AST'09. ICSE Workshop on*, pages 36–43. IEEE, May 2009.

[130] 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.

[131] 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.

[132] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.