# Alive2: Bounded Translation Validation for LLVM

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

Juneyoung Lee
juneyoung.lee@sf.snu.ac.k
Seoul National University
South Korea

Chung-Kil Hur
gil.hur@sf.snu.ac.k
Seoul National University
South Korea

Zhengyang Liu
liuz@cs.utah.edu
University of Utah
USA

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

## Abstract

We designed, implemented, and deployed Alive2: a *bounded* translation validation tool for the LLVM compiler's intermediate representation (IR). It limits resource consumption by, for example, unrolling loops up to some bound, which means there are circumstances in which it misses bugs. Alive2 is designed to avoid false alarms, is fully automatic through the use of an SMT solver, and requires no changes to LLVM. By running Alive2 over LLVM's unit test suite, we discovered and reported 47 new bugs, 28 of which have been fixed already. Moreover, our work has led to eight patches to the LLVM Language Reference—the definitive description of the semantics of its IR—and we have participated in numerous discussions with the goal of clarifying ambiguities and fixing errors in these semantics. Alive2 is open source and we also made it available on the web, where it has active users from the LLVM community.

***CCS Concepts:*** • **Software and its engineering → Software verification**; **Software verification and validation**; **Compilers**; **Semantics**; • **Theory of computation → Program verification**; **Program semantics**.

***Keywords:*** Translation Validation, Compilers, IR Semantics, Automatic Software Verification

**ACM Reference Format:**
Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454030

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*PLDI '21, June 20–25, 2021, Virtual, Canada*
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454030

## 1 Introduction

LLVM is a popular open-source compiler that is used by numerous frontends (e.g., C, C++, Fortran, Rust, Swift), and that generates high-quality code for a variety of target architectures. We want LLVM to be correct but, like any large code base, it contains bugs. Proving functional correctness of about 2.6 million lines of C++ is still impractical, but a weaker formal technique—translation validation—can be used to certify that individual executions of the compiler respected its specification.

A key feature of LLVM that makes it a suitable platform for translation validation is its intermediate representation (IR), which provides a common point of interaction between frontends, backends, and middle-end transformation passes. LLVM IR has a specification document,[1] making it more amenable to formal methods than are most other compiler IRs. Even so, there have been numerous instances of ambiguity in the specification, and there have also been (and still are) points of disagreement between the specification and the implementation. These disagreements span the spectrum from straightforward bugs all the way to fundamental flaws in the IR's design.

The technical core of our work is Alive2, the first fully automatic bounded translation validation tool for LLVM that supports all of its forms of undefined behavior (UB). Alive2 works with any intra-procedural optimization, and does not require any changes to the compiler. It checks pairs of functions in LLVM IR for *refinement*. A refinement relation is satisfied when, for every possible input state, a *target* function displays a subset of the behaviors of a *source* function. Refinement allows a transformation to remove non-determinism, but not to add it. In the absence of undefined behaviors, refinement degenerates to simple equivalence.

Handling undefined behavior (UB) is important because LLVM's optimizers frequently take advantage of it. UB is heavily used by frontends to communicate invariants about the code to optimizers. Therefore, in practice, any optimization verification tool that targets LLVM (or any other modern compiler) needs to support UB; otherwise the number of false alarms would make the tool impractical.

[1] https://llvm.org/docs/LangRef.html

Making Alive2 fully automatic necessitated some compromises. Our goal is a zero false-alarm rate, so we err on the side of soundness. We make use of *bounded* translation validation to bound the resources used by each verification task. For example, we unroll loops up to a given bound, and we limit execution time as well as memory consumption.

Alive2 has found, and continues to find, bugs in LLVM where optimization passes violate refinement. Some of these bugs stem from routine implementation errors, but the broader situation is much more interesting. LLVM is a large project that moves rapidly and has hundreds of people actively contributing to its code base. Given the highly informal English-language IR specification and the loose coordination among developers, it is inevitable that the lack of consensus on corner cases in the semantics leads to subtle compiler defects. For example, the LLVM IR specification was vague about the interaction between the `select` instruction (similar to C's ternary operator) and **poison** values (a form of UB). There are at least five possible semantics for this interaction and, using Alive2, we found that several of them were in active use in different parts of the compiler! This discrepancy led to end-to-end miscompilations in the wild. Fixing this kind of defect goes well beyond what would normally be considered formal methods research: we had to explain the problem to the LLVM developer community, present the different choices for the semantics, reach consensus on which one offered the best overall set of tradeoffs, and then help fix both the specification and the incorrect elements of the implementation. To be clear, we do not insist that LLVM conforms to our formalization of its semantics, but we do insist that it conforms to some semantics, and we have been iterating with the community in order to figure out what it should be.

Our work has emphasized transitioning formal methods tools and their results into the LLVM community, including:

- Creating several Alive2-based tools such as plugins for `opt` (LLVM's standalone IR optimizer) and Clang (LLVM's C/C++ frontend) that can be used to check refinement after every optimization pass, and a refinement checker for pairs of files containing LLVM IR.
- Since summer of 2019, continuously monitoring the LLVM unit test suite using Alive2, resulting in 47 bug reports, of which 28 have been fixed.
- Contributing 7 bug fixes to LLVM.
- Contributing 8 patches to LLVM IR's specification document and leading the discussion for fixing several other inconsistencies in the semantics.
- Running larger-scale experiments doing translation validation while compiling small applications.
- Engaging with the LLVM community on mailing lists and at their annual developers' meeting, where we have presented three talks about our work.
- Releasing Alive2 as open-source software (https://github.com/AliveToolkit/alive2) and also making it available through a web site (https://alive2.llvm.org/), obviating the need for most developers to compile our tools.

Our long-term goal is to collaborate with the LLVM community to iteratively improve the compiler implementation and the semantics of its IR, bringing the two into conformance while still meeting LLVM's other goals such as high-quality code generation. We also want to make Alive2 usable enough that LLVM developers can easily check candidate transformations for correctness before pushing them into the compiler.

## 2 Overview of LLVM

LLVM has a typed, SSA-based IR, supporting fixed bit-width integers, floats, pointers, vectors, arrays, and structures. Vectors are meant to be used in SIMD operations and only support indexing with constants, while arrays can be indexed with variables. Vectors are homogeneous whereas structures (which also only support indexing with constants) are heterogeneous.

LLVM's IR has two forms of undefined behavior (UB): immediate UB and deferred UB (**undef** and **poison** values) [24]. Immediate UB (or just "UB," for short) is the strongest form of undefined behavior and it is used for operations that trap on common CPUs, such as division by zero or dereferencing a null pointer. These operations cannot, in general, be speculated by compiler transformations, because this can make a program more undefined.

Deferred UB is used to define the semantics of operations for which the compiler either does not wish to impose a specific result, for performance reasons, or the operation is mathematically undefined. For example, shifting an integer by an amount not less than its bit-width produces different results on ARM and x86 CPUs, and therefore LLVM defines it as **poison**. Using **poison** rather than UB is important because it permits the compiler to speculatively execute potentially-undefined operations, for example by hoisting them out of loops or by flattening simple conditional code.

Deferred UB can turn into immediate UB when observed by certain operations. For example, branching on an **undef** or **poison** value is UB. In contrast, arithmetic operations simply propagate **poison** (akin to NaN in IEEE floating point).

Undef values are weaker than **poison**: they represent any value of their type and they do not taint operations like **poison**. A complication with **undef** values is that they may yield a different value each time they are observed. For example, **undef** & 1 can arbitrarily take the value 0 or 1 each time this expression is observed, while **poison** & 1 is **poison**. Undef values are mostly used to define the semantics of loading non-initialized memory.

Figure 1 illustrates some features of LLVM IR. This function takes two 32-bit integers as arguments, but there are actually $2^{2^{32}}$ possible values for each argument because each

```
define i32 @fn(i32 %a, i32 %b) {
  %t = add i32 %a, %a
  %c = icmp eq %t, 0
  br i1 %c, label %then, label %else

then:
  %q = shl i32 %a, 2
  ret i32 %q

else:
  %r = and i32 %b, 1
  ret i32 %r
}
```

**Figure 1.** Example LLVM IR function

may be **poison** or (perhaps partially) **undef**.[2] The first line of the function adds the first argument %a to itself. This is not equivalent to multiplying %a by two (resulting in an even number or **poison**) if %a is **undef**. Since **undef** may yield a different value each time it is observed, the two references to %a in the add instruction may not resolve to the same value and therefore the result of the addition could be odd.

The function then branches based on whether the result of the addition is zero or not. The branch instruction triggers UB if its input is either **undef** or **poison**. Therefore, the compiler can assume that from that point on %c is not **poison**, nor is %t or %a (as the integer comparison and integer arithmetic instructions propagate **poison**).

The then basic block returns %a left shifted by two, and because of the reasoning of the previous paragraph, we know that %q is not **poison**. The else basic block returns an expression based on %b. If %b is **undef**, the returned value is a partial **undef** value that can evaluate to zero or one.

Besides **undef** values, LLVM has one more form of nondeterminism through its **freeze** instruction. This instruction is used to control UB by stopping propagation of **undef** and **poison**. The result of **freeze** is a well-defined, arbitrarily chosen singleton value. Consider the following example:

```
%f = freeze i32 %a
%b = add i32 %f, %f
```

If %a is **undef**, %f will take an arbitrary, yet fixed value. Therefore, %b will always be an even number.

## 3 Encoding LLVM IR Semantics in SMT

In this section we explain how Alive2 encodes the state of an LLVM IR function and how the semantics of the IR are specified. Figure 2 defines values, the register file, and the program state. A program state consists of a register file, a memory, and a flag stating whether the program has executed UB. The program state is updated after the execution of each instruction.

[2]A large number of LLVM bugs have been caused by developers who neglected to consider the possibility that values flowing into a transformation they implemented could be undef or poison. The huge number of possible values for each argument is because partial **undef**s are effectively a powerset of its type's values.

| | |
|---|---|
| Num($sz$) | $::= \{ i \mid 0 \le i < 2^{sz} \}$ |
| BlockID | $::= \mathbb{N}$ |
| Offset | $::= \text{Num}(64)$ |
| Pointer | $::= \text{BlockID} \times \text{Offset}$ |
| DefinedValue | $::= \text{Int} \uplus \text{Pointer} \uplus \text{Float}$ |
| Value | $::= \mathcal{P}(\text{DefinedValue}) \uplus \{ \textbf{poison} \} \uplus \text{Aggregate}$ |
| Aggregate | $::= \text{list Value}$ |
| Memory | $::= \text{BlockID} \longrightarrow \text{MemBlock}$ |
| RegFile | $::= \text{string} \longrightarrow \text{Value}$ |
| State | $::= \text{RegFile} \times \text{Memory} \times \text{bool}$ |
| ValueNoRet | $::= \text{Value} \uplus \{ \text{noreturn} \}$ |
| FinalState | $::= \text{ValueNoRet} \times \text{Memory} \times \text{bool}$ |

**Figure 2.** Definitions. $\mathcal{P}$ is the power set operation.

($\iota$ = "$r$ = **add** i$sz$ $op_1$, $op_2$")

ADD-POISON

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

ADD

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

($\iota$ = "$r$ = **add nuw** i$sz$ $op_1$, $op_2$")

ADD-NUW-OVERFLOW

$$\frac{[\![op_1]\!]_R = v_1 \quad [\![op_2]\!]_R = v_2 \quad v_1, v_2 \in \mathcal{P}(\text{Int}) \qquad \exists i_1 \in v_1, i_2 \in v_2 \,.\, i_1 + i_2 \ge 2^{sz}}{\langle R, M, b\rangle \overset{\iota}{\hookrightarrow} \langle R[r \mapsto \textbf{poison}], M, b\rangle}$$

($\iota$ = "$r$ = **udiv** i$sz$ $op_1$, $op_2$")

UDIV-UB

$$\frac{[\![op_2]\!]_R = v_2 \qquad v_2 = \textbf{poison} \lor 0 \in v_2}{\langle R, M, b\rangle \overset{\iota}{\hookrightarrow} \langle R, M, \textbf{true}\rangle}$$

UDIV-POISON

$$\frac{[\![op_1]\!]_R = \textbf{poison} \quad [\![op_2]\!]_R = v_2 \qquad v_2 \in \mathcal{P}(\text{Int}) \land 0 \notin v_2}{\langle R, M, b\rangle \overset{\iota}{\hookrightarrow} \langle R[r \mapsto \textbf{poison}], M, b\rangle}$$

($\iota$ = "$r$ = **freeze** i$sz$ $op$")

FREEZE-POISON

$$\frac{[\![op]\!]_R = \textbf{poison} \qquad v \in \text{Num}(sz)}{\langle R, M, b\rangle \overset{\iota}{\hookrightarrow} \langle R[r \mapsto \{v\}], M, b\rangle}$$

FREEZE-PICK

$$\frac{[\![op]\!]_R = v \quad v' \in v \qquad v \in \mathcal{P}(\text{DefinedValue})}{\langle R, M, b\rangle \overset{\iota}{\hookrightarrow} \langle R[r \mapsto \{v'\}], M, b\rangle}$$

**Figure 3.** Semantics of selected instructions

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

We give the semantics for a few example instructions in Figure 3. Let tuple $S = \langle R, M, b\rangle$ be a program state (resp.

register file, memory, UB flag). The notation $S \xhookrightarrow{\iota} S'$ defines the resulting state S' of executing instruction $\iota$ on state $S$.

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

### 3.1 Register File

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

### 3.2 Function Arguments

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

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

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

### 3.3 Undef Values

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

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

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

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

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

Variables $\mathsf{undef}_i$ are appropriately quantified so they can take any value of the type. We describe this process later in Section 5.

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

Detecting **undef** boils down to detecting if an expression can evaluate to more than one value. The straightforward way for checking this for an expression $e$ with the set of undef variables $v$ is to check if $\exists v, w \,.\, e \neq e[w/v]$ is satisfiable (with $w$ being a set of fresh variables). To reduce the number of variables, we use an alternative encoding: we replace variables $v$ with a constant, $e_k = e[k/v]$. If the comparison $e = e_k$ is valid, then $e$ is not **undef** since there is no model for the undef variables that makes $e$ different than a base value ($e_k$). We tried the values 0 and 1 for this constant, but neither worked well because they are identity/absorbent for some arithmetic operations and are folded away by the solver. This tends to destroy syntactic similarity between $e$ and $e_k$, confusing our SMT solver's quantifier instantiation algorithm. The constant 2 also does not work well: it is a power of 2 and thus simplifies, e.g., multiplications. Therefore, we replace **undef** values with 3 when creating $e_k$.

### 3.4 Control Flow

The flow of execution in LLVM is controlled using (conditional) branch, `switch`, function calls (Section 6), and exceptions (`invoke` instruction). Support for loops in Alive2 is described in Section 7.

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

### 3.5 Floating-Point Numbers

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

types such as x86's 80-bit floats, as SMT's FPA theory does not support those either.

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

There are essentially two choices:

- `bitcast` from integer to float and then back to integer preserves the bit pattern (such a round-trip is a NOP).
- `bitcast` does not preserve NaN's bit pattern. When a NaN is bit casted to integer, it gets assigned a non-deterministic bit pattern.

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

### 3.6 Return Value

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

The SMT encoding for the register file of the function shown in Figure 1 is:

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

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

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

The final state for the same function is as follows:

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

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

### 3.7 Additional Optimizations

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

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

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

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

### 3.8 Dealing with Unsupported LLVM Features

In its role as a robust IR supporting a number of source languages, a number of target architectures, and aggressive optimization, LLVM IR has accrued many features. Alive2 does not handle them all; we have not yet supported exceptions (the `invoke` instruction), function pointers, volatile variables, pointer-to-integer casts, type-based alias analysis,[3] and many of LLVM's intrinsics, which are essentially non-core IR instructions. Out of the 258 non-experimental platform-independent intrinsics, we support 54 (21%).

[3] No formal model of type-based aliasing exists, so far.

Beyond intrinsics, LLVM has coarse-grained semantics for 463 library functions including some from the C library, C++ library, Objective C runtime, etc. These specifications include predicates such as: the function always returns a non-null pointer, the function always/never returns, the function only reads/modifies objects referenced by the arguments, and so on. Since LLVM optimization passes take advantage of these predicates to limit the scope of behavior of function calls, Alive2 has to mirror this knowledge. For example, LLVM transforms a `printf()` of a constant string into a call to `puts()`; this looks like a failure of refinement unless we equip Alive2 with analogous knowledge about this pair of functions. So far we have special-cased the semantics of 117 library functions, some of which only partially.

A major design goal of Alive2 is to avoid false alarms while attempting to verify as much as possible. Therefore, when it encounters an unsupported feature, Alive2 attempts to produce an over-approximation of the semantics of that feature. For example, if we encounter a call to an unsupported intrinsic, we encode it as a call to an unknown function, which can potentially modify all the memory and return an arbitrary value. To avoid false alarms, we tag the function call as being an over-approximation. If later on Alive2 finds a bug in a transformation, we check whether an over-approximation was involved. To this end, we record all SMT expressions produced when encoding over-approximated features. Since the SMT solver we use produces partial models (i.e., it may leave some variables unassigned when it declares a formula satisfiable if those variables can take any value without making the formula unsatisfiable), we check if any expression from over-approximated features is in the model. If not, it means the bug Alive2 found is real as it does not depend on the specifics of an over-approximated feature. Otherwise, if some expression is in the model, we cannot conclude anything: it may or may not be a bug in LLVM. In this case, we do not report the transformation as incorrect, but instead list the over-approximated features that prevented Alive2 from verifying the transformation.

Some features are not easily over-approximated, such as function pointers (unsupported at the moment). We skip functions containing any of such features.

## 4 Encoding Memory in SMT

We briefly describe our SMT encoding of LLVM's memory model [21]. [4] In this paper, we only consider logical pointers (i.e., integer-to-pointer casts are not supported) and a single address space.

***Memory blocks.*** The unit of memory allocation is the memory block. Each stack/global variable gets a distinct block, and calls to **malloc** create a fresh block. Each block is uniquely identified with a non-negative integer bid.

[4] A more detailed description is given in [23].

After loops are unrolled, we statically compute the maximum number of memory blocks a program can touch so we can bound the number of bits needed to encode bid. We need to take into account the number of pointer inputs, stack and global variables, and instructions that may return a fresh pointer (e.g., function calls that return a pointer).

***Pointers.*** A pointer is defined as a pair $(\mathrm{bid}, \mathrm{off})$ which is encoded as a bit-vector by concatenating bid and off (SMT variables if the pointer may alias multiple blocks). null is defined as $(0, 0)$ pointing to a null block having size 0. **undef** pointers are defined as $(\beta, \omega)$ where $\beta, \omega$ are fresh variables.

The pointer arithmetic instruction (**gep** $ptr,\ i$) returns a new pointer with $i$ added to $ptr$'s offset (bid is unchanged). If the **inbounds** attribute is present, the result is **poison** if either the base or the resulting pointers are out-of-bounds.

***Block attributes and bytes.*** Each block has associated properties such as size, alignment, whether it is read-only, whether it is alive, allocation type, and physical address.

To encode a memory block's value, we use an SMT array from pointer to byte. Bytes have three possible types: poison, pointer, and non-pointer. A non-pointer byte uses an 8-bit bit-vector for the value, as well as an 8-bit mask to record which bits are poison. Floats are converted to bit-vectors for storage.

***Memory accesses.*** A possibly multi-byte load/store is split into single-byte loads/stores. These are then combined to produce a value of appropriate type (and size, as types' sizes are not necessarily multiples of bytes).

The result of a load is **poison** if any of the following holds: (1) any of the loaded bytes is poison, (2) some of the loaded bytes have different types, (3) the load type does not match the stored byte's type (e.g., attempt to load a pointer from a block that has an integer stored).

We keep track of the undef variables used in stored values and pointers. These have to be replaced with fresh variables in each loaded value. Memory accesses using out-of-bounds pointers or to already-freed blocks trigger UB. Store operations also trigger UB if the block is read-only.

The **free**$(ptr)$ operation updates the liveness of the corresponding block. It triggers UB if the block is already dead or not allocated on the heap, or if the offset is not zero.

## 5 Verifying Correctness of Optimizations

To verify correctness of LLVM optimizations, we establish a refinement relation between source (original) and target (optimized) code. We cannot simply check for equivalence because UB-related transformations are ubiquitous.

Given functions $f_{src}$ and $f_{tgt}$, a set of input and output variables $I_{src}/I_{tgt}$ and $O$ (which include, e.g., memory, side effects, and the return value), a set of non-determinism variables

ELEMENT-NONPTR
$$\frac{v \in \text{Int} \uplus \text{Float}}{v \sqsupseteq_e v}$$

ELEMENT-PTR
$$\frac{v, v' \in \text{Pointer} \quad \text{pointerRefined}(v, v')}{v \sqsupseteq_e v'}$$

VALUE-POISON
$$\frac{v \in \text{Value}}{\textbf{poison} \sqsupseteq v}$$

VALUE-NORETURN
$$\frac{v = \text{noreturn}}{v \sqsupseteq v}$$

VALUE-UNDEF
$$\frac{v \in \mathcal{P}(\text{DefinedValue}) \quad v' \in \mathcal{P}(\text{DefinedValue}) \quad \forall v' \in v' \,.\, \exists v \in v \,.\, v \sqsupseteq_e v'}{v \sqsupseteq v'}$$

VALUE-AGGREGATE
$$\frac{v, v' \in \text{Aggregate} \quad |v| = |v'| \quad \forall i \,.\, v[i] \sqsupseteq v'[i]}{v \sqsupseteq v'}$$

FINAL-STATE-UB
$$\frac{r, r' \in \text{ValueNoRet} \quad M, M' \in \text{Memory} \quad ub' \in \text{bool}}{\langle r, M, \textbf{true} \rangle \sqsupseteq_{st} \langle r', M', ub' \rangle}$$

FINAL-STATE
$$\frac{r, r' \in \text{ValueNoRet} \quad M, M' \in \text{Memory} \quad r \sqsupseteq r' \quad M \sqsupseteq_m M'}{\langle r, M, \textbf{false} \rangle \sqsupseteq_{st} \langle r', M', \textbf{false} \rangle}$$

**Figure 4.** Refinement of value and final state

$N_{src}/N_{tgt}$, $f_{src}$ is refined by $f_{tgt}$ iff:

$$\forall I_{src}, I_{tgt}, O \,.\, \left(I_{src} \sqsupseteq I_{tgt} \;\wedge\; \exists N_{tgt} \,.\, f_{tgt}(I_{tgt}, N_{tgt}, O)\right) \implies (\exists N_{src} \,.\, f_{src}(I_{src}, N_{src}, O))$$

In other words, for any fixed input $I_{src}$, if the target function $f_{tgt}$ produces a given output $O$ with some internal non-determinism $N_{tgt}$ and refined input $I_{tgt}$ (equal to or more defined than $I_{src}$), the source function must produce the same output for some internal non-determinism $N_{src}$. Therefore, the target function is allowed to remove non-determinism so it generates fewer outputs for a given input, but not the other way around.

We only support intraprocedural optimizations, and therefore checking refinement of each function individually is sufficient to establish refinement of an entire program. The definition above ensures compositionality: if a function's inputs are refined, so are the outputs. This definition is assumed at call sites, and established for each function, justifying why checking each function individually is sufficient.

In LLVM, loops tagged with the **mustprogress** attribute must either terminate or perform externally observable actions infinitely often. The function triggers UB otherwise. Therefore, every function in LLVM with only such loops terminates, perhaps triggering UB. In practice, the compiler can only prove non-termination of simple cases and therefore we only need to support those. Moreover, given that we do *bounded* translation validation, we do not support non-terminating loops without the **mustprogress** attribute.

### 5.1 Refinement of Program State

We start by defining refinement between values. A value $v$ is refined by another value $v'$, or $v \sqsupseteq v'$, if $v'$ is equivalent to or more defined than $v$. Figure 4 gives rules for the definition of $v \sqsupseteq v'$.

For integer and floating-point numbers, refinement holds between two equal numbers (ELEMENT-NONPTR). For pointers, we cannot simply use equality because local pointers (such as pointers to stack-allocated memory blocks) in source and target are internal to each of the functions. Therefore, we use a function that compares local and non-local pointers appropriately (ELEMENT-PTR). **poison** is refined by any value (VALUE-POISON). noreturn is refined by itself only (VALUE-NORETURN). A (partially) undef value is refined by another undef value that is equally or more defined (VALUE-UNDEF). Finally, aggregate values are compared element-wise (VALUE-AGGREGATE).

Next, we define refinement between final states. A final state is defined as a tuple $\langle r, M, ub \rangle$, where $r$ is the return value, $M$ the memory at the return site, and $ub$ a Boolean flag indicating whether the function triggered UB before returning. A memory $M$ is refined by $M'$, $M \sqsupseteq_m M'$, if refinement holds between blocks in $M$ and $M'$ (value and remaining attributes). A final state $s$ is refined by $s'$, or $s \sqsupseteq_{st} s'$, if (1) $s$ is undefined (FINAL-STATE-UB), or (2) refinement between their respective return values and memories holds (FINAL-STATE).

Using $\sqsupseteq_{st}$, we define correctness of an optimization as follows. If functions $f_{src}$ and $f_{tgt}$ are deterministic, $f_{src}$ is refined by $f_{tgt}$ if:

$$\forall I_{src}, I_{tgt} \,.\, I_{src} \sqsupseteq I_{tgt} \;\wedge\; \text{valid}(I_{src}, I_{tgt}) \implies [\![ f_{src} ]\!](I_{src}) \sqsupseteq_{st} [\![ f_{tgt} ]\!](I_{tgt})$$

The valid predicate encodes the global precondition. For example, it states that global variables should be assigned non-null and disjoint addresses. $[\![ f ]\!](I)$ is the final state after executing function $f$ with input $I$.

### 5.2 Nondeterministic Execution

The previous definition of correctness does not take non-determinism, such as **undef** values and **freeze** instructions, into account. Let $N_{src}$ and $N_{tgt}$ be the set of variables used to encode non-determinism in functions $f_{src}$ and $f_{tgt}$, respectively. We extend the previous definition of refinement to support non-determinism as follows:

$$\begin{aligned} &\forall I_{src}, I_{tgt}, O_{tgt} \,.\, I_{src} \sqsupseteq I_{tgt} \;\wedge\; \text{valid}(I_{src}, I_{tgt}) \;\wedge \\ &\quad (\exists N_{tgt} \,.\, \text{pre}_{\text{tgt}}(I_{tgt}, N_{tgt}) \;\wedge\; [\![ f_{tgt} ]\!](I_{tgt}, N_{tgt}) = O_{tgt}) \\ &\implies (\exists N_{src} \,.\, \text{pre}_{\text{src}}(I_{src}, N_{src}) \;\wedge\; [\![ f_{src} ]\!](I_{src}, N_{src}) \sqsupseteq_{st} O_{tgt}) \end{aligned}$$

Predicate pre represents the precondition of a function, which is used to constrain the non-determinism and the inputs a function can take. For example, in LLVM a function's argument can be marked as non-null. Constraints like this are added to pre.

Sometimes the precondition for the source function does not hold for a particular input $I_{src}$ for any non-determinism but it may hold for the target function. For example, LLVM is allowed to remove the non-null attribute of a function's argument. The formula above fails in that case, since then $\text{pre}_{\text{src}}$ makes the right-hand side of the implication become false.

CALL

$$\frac{\begin{array}{ll} args, args' \in \text{list Value} & (M_o, v_o, ub_o) = \texttt{call}(fn,\ args,\ M) \\ M, M' \in \text{Memory} & (M'_o, v'_o, ub_{o'}) = \texttt{call}(fn,\ args',\ M') \\ M \sqsupseteq_m M' & \forall i\ .\ args[i] \sqsupseteq_{\text{arg}} args'[i] \end{array}}{M_o \sqsupseteq_m M'_o \ \wedge\ v_o \sqsupseteq v'_o \ \wedge\ (ub'_o \implies ub_o)}$$

**Figure 5.** Refinement between the inputs and outputs of two function calls

To support such cases, we extend the previous refinement condition to arrive at the final version that Alive2 actually uses:

$$\begin{aligned} &\forall I_{src},\ \ I_{tgt}, O_{tgt}\ .\ \text{valid}(I_{src}, I_{tgt}) \wedge \\ &\qquad (\exists N_{src},\ N_{tgt}\ .\ \text{pre}_{\text{src}}(I_{src}, N_{src}) \ \wedge\ \text{pre}_{\text{tgt}}(I_{tgt}, N_{tgt}) \wedge \\ &\qquad\qquad [\![ f_{tgt} ]\!](I_{tgt}, N_{tgt}) = O_{tgt}) \\ &\implies \ (\exists N_{src}\ .\ \text{pre}_{\text{src}}(I_{src}, N_{src}) \ \wedge\ [\![ f_{src} ]\!](I_{src}, N_{src}) \sqsupseteq_{st} O_{tgt}) \end{aligned}$$

### 5.3 SMT Encoding

To check refinement using an SMT solver, the last formula from the previous subsection is negated and the SMT solver is asked to prove unsatisfiability. Rather than running a monolithic query, we check refinement as a sequence of simpler queries; this helps provide detailed error messages to users and also reduces the burden on the solver. We check if:

1. Any of the preconditions is always false; this can happen because of bugs or limitations in the encoding.
2. The target triggers UB only when the source does.
3. The return domain of the target is equal to that of the source, except for when the source triggers UB.
4. The return value of the target is **poison** only when the source's return value is **poison**.
5. The return value of the target is **undef** only when the source's return value is **undef** or **poison**.
6. The return value of the source and target are equal when the source value is neither **undef** nor **poison**.
7. Finally, if memory is refined.

## 6 Function Calls

A call to an unknown function may change the memory in an arbitrary way. We model the semantics of call instructions as a pure function that takes its arguments as well as the current memory as inputs, and returns a value and a fresh memory. Let $(M_o, v_o, ub_o) = \texttt{call}(f,\ args,\ M)$ denote a call.

***Relating two function calls.*** When considering multiple calls to the same function, we may need to relate the impact each has on the program state. For the purpose of refinement checking, there are three cases to consider: (1) two calls in source, (2) two calls in target, (3) a call in source and another in target.

For the first case, we consider all pairs of calls in the source to a same function and constrain their behavior such that their outputs are refined if the inputs are refined (c.f. Figure 5). We need to consider this case because LLVM has an optimization that removes duplicated calls. For example, if a function is known to not read or write to memory and it is called twice with the same arguments and one call dominates the other, LLVM removes the dominated call. In practice, LLVM's optimizer only de-duplicates calls with equal inputs (rather than refined), so similarly we make the condition for input refinement stronger (for performance reasons) without introducing false alarms.

Relating two calls in the target is not necessary: these calls must already exist in source (with potentially less defined inputs) and therefore they are already appropriately related, or else refinement does not hold as it is illegal to introduce new function calls. Relating calls in source and target is similar to the first case, but we need to use the full refinement rule as shown in Figure 5.

***SMT encoding.*** Each call in the source function gets a fresh variable for each of its outputs (memory, return value, and UB flag). Calls in the target are encoded differently, as these must refine at least one of the source's calls since no new calls can be introduced.

Focusing just on poison, a value $v$ is refined by $v'$ iff $v$ is **poison** or $v = v'$. Same reasoning applies for **undef**. Therefore, we consider the two parts of the encoding differently: (1) the poison flag, and (2) the value (only meaningful when the poison flag is false). For the poison flag, we introduce a fresh variable for each call in the target. For the value part, we have to pick one of the source call's values or use a fresh one in case the source's value is poison.

Given that the non-deterministic variables of source and target are bound to different quantifiers in the refinement query, it is not trivial to encode the value part. Let $C^f_{src}$ be the set of calls to function $f$ in the source. We introduce a fresh variable $i$ such that $0 \leq i \leq |C^f_{src}|$. Then, in the precondition, we encode that $i = j$ holds iff the $j$th call in $|C^f_{src}|$ is refined by the target's call, except for $i = |C^f_{src}|$ that holds iff no source's call is refined. The target's value is encoded with an ite expression that ranges over $i$ and yields the corresponding source's value. The target triggers UB if $i = |C^f_{src}|$, i.e., when the call does not correspond to any of the source's.

A limitation of our current implementation is that local variables are never modified by function calls even if their address has escaped.

***Optimizations.*** The number of function call pairs that may need to be related grows quadratically with the number of calls to the same function. We use a dataflow analysis to cheaply prune call pairs that definitely do not have their inputs refined. For each call, we compute (in a path-sensitive way) the min/max number of calls of each function that were made in all preceding paths. Then we only consider call pairs with overlapping ranges. If ranges do not overlap, one of the calls must have had at least one more call beforehand which could have changed the memory that is read by the

function under consideration. Therefore, this is a sound over-approximation of possibly-related call pairs.

# 7 Loops

Alive2 performs *bounded* translation validation by unrolling loops in the source and target functions by a specified factor. Thus, it will find any failure of refinement that is triggered within the specified bound, and will miss refinement failures that require more loop iterations to manifest.

Alive2 implements Tarjan-Havlak's loop analysis algorithm [14] for recognizing the loops in a function and their nesting relation. The result of the analysis is a loop nesting forest (a collection of trees). In a nesting tree, each node represents a loop header and its children are the headers of the immediately nested loops.

Alive2 unrolls loops inside-out by traversing each loop nesting tree in post-order with a DFS. An advantage of this order is that the number of unrolls is linear in the number of loops and unroll factor instead of being exponential if done in the reverse order.

Alive2 unrolls a loop by repeatedly duplicating the loop until the unroll factor is reached. After duplication, three things need fixing: (1) instruction operands, (2) targets of jump instructions, and (3) introduce/patch $\phi$ nodes.

Operands are patched during basic block duplication. Alive2 maintains a map that records the duplicates of each of the original SSA values. When an instruction is duplicated, its operands are replaced with the latest duplicate in this map.

Jump targets are patched by replacing each target with its next duplicate. If no such basic block (BB) exists, it means it is a backedge in the last unroll. We redirect these jumps to a special sink BB. The reachability domain of the sink BB is negated and added to the precondition of its respective function. Since the amount of unrolled computation of the source/target functions may not be synchronized (e.g., for loop-manipulating optimizations), we need to restrict refinement checking such that the control flow cannot reach any of the sink BBs. The usage of sink BBs allows us to avoid false positives, but it prevents us from supporting infinite loops since we can only check refinement of paths that reach a `return` instruction.

Some instructions inside a loop may have their result used outside the loop. We need to patch such users so they observe the right values depending on how many loop iterations are executed. We implemented a conservative solution with three cases: patch existing $\phi$ nodes (just add more predecessors), introduce a new $\phi$ when the loop has a single exit to a BB that dominates [7] the user's BB, and otherwise fallback to using memory. Complex cases may require introducing several $\phi$ nodes [3]; we introduce a new stack variable to avoid having to maintain the SSA form altogether.

Picking the right unroll factor involves a tradeoff between coverage and run time. We note that the unroll factor should be at least two for optimizations that do not manipulate loops so we can cover the backedge entry in $\phi$ nodes. For loop-manipulating optimizations, this may have to go as high as 64, depending on the optimization. Vectorization may optimize, e.g., 32 iterations of the source loop into a single (vectorized) iteration, hence we need to unroll the source loop at least 64 times so it covers two iterations of the target loop.

# 8 Implementation and Evaluation

This section describes the implementation of Alive2, and evaluates its utility and its impact on the LLVM compiler and community.

## 8.1 Implementation

Alive2 consists in about 23,000 lines of C++ and uses Z3 [12] for SMT solving. Besides its own source code, the trusted computing base for Alive2 includes Z3 and functionality from LLVM for parsing binary and textual LLVM IR into an in-memory representation. Because a premise of our project is that we do not trust LLVM to be correct, Alive2 does not rely on code from LLVM to perform tasks such as computing points-to sets or dominator trees.

***Tools.*** Alive2 includes multiple tools for different use cases:

- A plugin for `clang` (LLVM's C/C++ frontend) that validates all optimizations performed by LLVM.
- A plugin for `opt` (LLVM's standalone optimization tool). We add a `-tv` argument so the user can choose after which optimizations Alive2 should run (to support batching), e.g., `opt -tv -sroa -instcombine -tv -gvn -tv file.ll`.
- `alive-tv`, a standalone tool that takes two LLVM IR files and checks refinement between each function present in the two files.
- A pair of compiler drivers, `alivecc` and `alive++`, that respectively invoke `clang` and `clang++` with options that cause our translation validation plugin to be loaded, and then to validate every intra-procedural IR-level transformation that the compiler performs.

Our LLVM plugins implement a trivial (but effective) additional optimization beyond those described earlier in this paper, which is to avoid running Alive2 at all when an LLVM pass does not make any changes.

## 8.2 Translation Validation of LLVM's Unit Tests

As of version 11, the LLVM test suite contains around 168,000 functions in LLVM IR that are variously optimized, analyzed, and compiled to machine code during testing. About 36,000 of these functions test IR-level transformations and this subset has been the focus of our translation validation efforts.

***LLVM's unit test framework.*** This is a typical test case:

```
; RUN: opt < %s -instsimplify -S | FileCheck %s

define i1 @max1(i32 %x, i32 %y) {
; CHECK-LABEL: @max1(
; CHECK:         ret i1 false
;
  %c = icmp sgt i32 %x, %y
  %m = select i1 %c, i32 %x, i32 %y
  %r = icmp slt i32 %m, %x
  ret i1 %r
}
```

It ensures that the instruction simplifier—a collection of peephole optimizations—can recognize that the maximum of two integer values cannot be smaller than the first of those values. The first line specifies that the `opt` tool should run the instruction simplifier and pipe its output through the `FileCheck` tool, which fails the test unless the function is optimized to return false.

To run one or more of these test cases through Alive2, we ask `lit`, the LLVM unit test runner, to call a program that we wrote, instead of calling `opt`. This program runs `opt` with our plugin and skips unsupported transformations (e.g., interprocedural optimizations). Our plugin then works in three stages. First, it translates the original LLVM IR into Alive2 IR and stores it in memory. Second, it runs the specified (unmodified) LLVM transformations (managed by LLVM's pass manager itself). Finally, it translates the optimized LLVM code into Alive2 IR and checks if it refines the original IR that was saved earlier. Thus, the LLVM unit tests can be run seamlessly through Alive2.

***Results.*** We detected 121 violations of refinement in the unit tests:

- 43 optimizations that are incorrect when **undef** is given as input or constant
- 18 optimizations that introduce a branch on **undef** or **poison**, which is UB
- 9 bugs due to the mishandling of vector operations
- 5 UB-related bugs while optimizing `select` instructions
- 4 incorrect arithmetic operations
- 4 loop optimizations incorrectly handling memory accesses
- 3 occurrences of incorrect handling of floating point "fast-math" flags
- 3 bugs due to the ambiguous semantics of bitcast between integer and floating points
- 17 memory-related miscompilations
- 15 failures due to bugs in Alive2, or tests that are designed to fail when an external module like Alive2 is invoked

We reported 47 miscompilation bugs to the LLVM community after identifying the root causes of unit test failures. We did not report every bug detected by Alive2 as some were

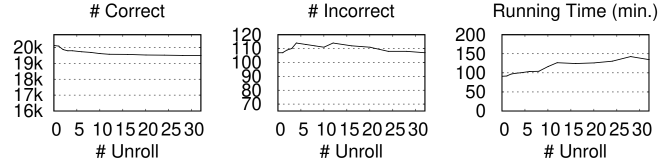


**Figure 6.** Effect of changing the unroll factor when validating LLVM's unit tests.

already known. At the time of publication, 28 of the bugs we reported have been fixed, including 7 patches that we wrote ourselves. The remaining fixes were done by LLVM developers, and we actively led discussions around finding good solutions for the bugs. In several cases, compiler developers used Alive2 to help validate that their fixes were correct. Moreover, several members of the LLVM community have become Alive2 users, and have gone on to fix LLVM bugs detected by Alive2, even though we never reported them.

***Selected bug #1: Vectorization.*** Here `%x` is a pointer to an array of 8-bit integers:

```
%a = load i8* %x
%b = load i8* (%x+1)
%c = load i8* (%x+2)
%d = load i8* (%x+3)    ⇒   %v = load <4 x i8>* %x
%r = %a +nsw %b +nsw        %w = %v[0:1] +nsw %v[2:3]
     %c +nsw %d             %r = %w[0]   +nsw %w[1]
```

This transformation, which exploits the associativity of addition to reduce the number of instructions using vector addition, is not a refinement. The problem is that LLVM's addition operator, when qualified by the `nsw` flag (which turns signed overflows into poison values), is not associative. The fix was to drop the `nsw` flag from the code on the target side of this transformation.

***Selected bug #2: Floating point.*** This transformation is not a refinement:

```
%c = fmul nsz %a, %b      %c = fmul nsz %a, %b
%r = fadd %c, +0.0    ⇒
ret %r                    ret %c
```

The `nsz` (non-signed-zero) flag is an assertion that `%c` is nondeterministically +0.0 or −0.0 if $\%a \times \%b = 0$. However, `%r` is +0.0 even if $\%c = -0.0$ due to the definition of floating point addition. Thus, the target code displays a behavior not observed in the source, violating refinement.

***Performance.*** Running the LLVM unit test suite under Alive2 takes about 2.5 hours on an 8-core Intel workstation. Figure 6 shows the trend of the number of passed tests, refinement failures, and running time when increasing the unroll factor. The number of passed tests decreases with the unroll factor due to timeout or out-of-memory. The wall-clock time increases in a linear manner.

***Discussion.*** One might wonder why the LLVM unit tests—which seldom fail on the main LLVM development branch—would be a fruitful place to look for compiler bugs. The answer is that Alive2 is a far more discerning test oracle than are the syntactic oracles, such as the `CHECK:` line in the example at the start of this subsection, that are built into the unit tests. Moreover, Alive2 has the virtue of being consistent: it expects all test cases to follow the same rules.

### 8.3 Updates to the LLVM IR Semantics

When we found ambiguities in the LLVM Language Reference, we initiated discussions in order to clarify the document. Overall, we wrote 5 patches and contributed advice or ideas to 3 patches written by LLVM developers. Here are some specific examples.

***GEP.*** When we started our work, it was not clear whether LLVM's **gep inbounds** operator for pointer arithmetic interpreted its index argument and the base pointer's offset value as signed or unsigned integers, for purposes of computing "inboundedness" of the resulting address. Also, the assumptions that an object cannot be larger than half of the size of its address space, and that no "inbounds" address computation can overflow an unsigned value, needed to be clarified.

***Branches and UB.*** In the past, we proposed that branching on an **undef** value should be UB [24], but the eventual adoption of this idea as the official LLVM semantics was guided by Alive2. This semantics justifies optimizations that rely on branch conditions, but makes it illegal to introduce new conditional branches in many situations. Alive2 found that LLVM was doing this kind of (now unambiguously incorrect) optimization.

***Vectors and UB.*** LLVM's **shufflevector** instruction supports permuting two input vectors, returning an output vector that has the same number of elements as its mask argument:

```
; Shuffles two vectors with mask <3, 2, 1, 2>
%v = shufflevector <10, 20>, <30, 40>, <3, 2, 1, 2>
; result: %v = <40, 30, 20, 30>
```

Initially, we believed that when the mask operand contained one or more **undef** values, **poison** elements in the input vectors would be propagated to the output. We reported optimizations that were incorrect under these semantics, and this led to discussions with LLVM developers followed by a decision that **undef** in the mask operand does not result in the propagation of **poison** values.

***Other changes.*** We helped make several clarifications regarding the interaction between **undef** and padding in aggregates. For example, **freeze** has no effect on padding values. Additionally, we clarified that a pointer given to a load or store instruction is not allowed to be a non-deterministic value.

| Prog. | LoC | Pairs | Diff | Time | ✓ | ✗ | TO | OOM | Unsup. |
|---|---|---|---|---|---|---|---|---|---|
| bzip2 | 5.1K | 282K | 2.2K | 1.26 | 333 | 10 | 540 | 195 | 1,125 |
| gzip | 5.3K | 371K | 2.6K | 1.74 | 884 | 4 | 905 | 60 | 754 |
| oggenc | 48K | 215K | 1.8K | 1.63 | 440 | 4 | 588 | 72 | 663 |
| ph7 | 43K | 1.7M | 5.6K | 3.15 | 1,393 | 28 | 1,337 | 35 | 2,755 |
| SQLite3 | 141K | 3.9M | 12.2K | 6.37 | 2,314 | 38 | 2,102 | 100 | 7,543 |

**Figure 7.** Results for single-file benchmarks. From left to right, the columns indicate the program name, the number of lines of code, the total number of source/target function pairs (intraprocedural optimizations only), the number of non-identical pairs considered for translation validation, total wall-clock time taken (in hours), pairs successfully validated, violations of refinement, timeouts, out-of-memory conditions, and pairs containing at least one feature unsupported by Alive2.

### 8.4 Translation Validation for Applications

Although our focus has been on validating transformations for core elements of LLVM IR, we also wanted to see how Alive2 would work while compiling applications. We chose five single-file benchmarks: bzip2, gzip, oggenc,[5] ph7 2.1.4,[6] and SQLite 3.30.1 amalgamation,[7] and compiled them at the `-O3` optimization level, and using the `-fno-strict-aliasing` flag to disable type-based alias analysis. We extracted pairs of IR files corresponding to the code before and after every optimization pass ran on every function in the code being compiled. We timed out individual invocations of Z3 after one minute and limited its RAM usage to 1 GB.

We batched optimization passes for oggenc, ph7, and SQLite, in order to reduce the total verification time. Instead of calling Alive2 after each optimization, we batched optimizations between pairs of unsupported optimizations, such that only supported transformations occurred between those two optimizations. Batching, however, incurs a slight risk of hiding bugs, as an optimization may accidentally fix the miscompilation of a previous optimization.

The results of this experiment, run on an 8-core machine, are shown in Figure 7. Quite a few functions in these programs make use of features not yet supported by Alive2; fixing these is a matter of ongoing work. The most common unsupported features are function pointers and missing semantics for some string and I/O library functions. Furthermore, LLVM IR has a long tail of features that have been added over the years, and supporting them requires significant engineering effort.

The last five columns almost add up to the "Diff" column, which is the number of function pairs for which we run Alive2. The remaining few pairs not shown in the table could not be proved correct or incorrect due to Z3 giving up because of incomplete handling of quantifiers.

[5] http://people.csail.mit.edu/smcc/projects/single-file-programs
[6] http://www.symisc.net/downloads/ph7-amalgamation-2001004.zip
[7] https://www.sqlite.org/2019/sqlite-amalgamation-3300100.zip

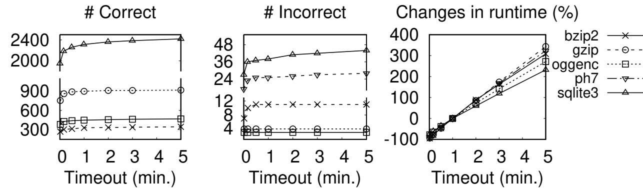


**Figure 8.** Effect of changing the SMT solver timeout for the single-file benchmarks. In the third graph, at the right, running times are normalized to the time taken by that benchmark when the timeout is set to one minute.

We manually inspected every failure of refinement observed during this experiment. The bulk of them are due to an incorrect transformation done by LLVM, where in some cases **select** instructions with Boolean operands are replaced with **and**/**or** instructions. For example, LLVM currently replaces **select** %x, %y, **false** with **and** %x, %y. This transformation is incorrect if %y may be **poison** as the **select** instruction would yield **false** if %x = **false** (short-circuiting semantics), while the **and** yields **poison**. A fix is ongoing and has required fixing several optimizations to accept the new canonicalization, as well as fixing the backend (SelDAG) to produce good code for Boolean **select** instructions.

If this experiment had been the first thing we did using Alive2 (as opposed to spending more than a year iteratively improving LLVM by reporting and fixing bugs found by running Alive2 on the unit test suite) then we would have found many more refinement failures. Of course our long-term goal is to fix LLVM until Alive2 finds no failures.

***Measuring the effect of SMT solver timeout.*** To better understand the impact of the SMT solver's timeout, we ran the single-file benchmarks with timeouts varying from one second to five minutes; the results are shown in Figure 8. While the running time of Alive2 increases approximately linearly with the solver timeout, the number of times Alive2 reached a definitive result plateaus once the timeout reaches one minute. Increasing the solver timeout from one to five minutes increased the number of pairs proved correct and incorrect by less than 5% and 17%, respectively.

### 8.5 Reproducing Known LLVM Bugs

To test whether Alive2 misses bugs in practice, we used it to analyze some incorrect intra-procedural transformations that were reported publicly, but not by us. We investigated 36 bug reports picked at random. 29 of these refinement failures were detected by Alive2 and the remaining seven were missed. Of the missed alarms, one was due to the existence of an infinite loop (unsupported), one was due to the required unroll factor being too large (the loop requires about $2^{16}$ iterations to exit), and five were because our memory encoding assumes that function calls do not modify escaped stack variables. Full support for LLVM's memory model is out of scope of this paper. We manually changed the tests to have loops with fewer iterations or to escape local variables to globals, and confirmed that Alive2 could find all bugs except for one that fails to validate within a one-hour timeout.

### 8.6 Z3 Bugs Found While Developing Alive2

Although finding defects in Z3 was not one of our goals, we did encounter solver bugs while performing this work. We found six soundness bugs, six crashes, and one timeout violation. All but one of these bugs have been fixed; two by us, and the rest by the Z3 developers.

We also hit some performance issues in Z3. We found that one of Z3's internal timer mechanisms was incurring significant overhead because it created a new helper thread on every use of the timer. By patching Z3 to use a thread pool for its `scoped_timer` abstraction, we realized a 20–30% speedup for a collection of Alive2 processes running on a large multicore. We also found an issue in the structural hashing mechanism that was having too many collisions in the hash table. Fixing this issue led to a 3x speedup in the sqlite3 benchmark. Finally, we introduced a new API in Z3 to reset the solver's memory, to reduce memory fragmentation; this fixed an issue where the performance of long Alive2 runs degraded over time. All these patches have been upstreamed.

## 9 Related Work

***Translation validation.*** Early translation validation (TV) tools supported only transformations that did not change the control-flow or the loop structure (e.g., loop unroll, software pipeline) of the program [37]. TV tools were then extended to accept hints from the compiler (witnesses) to simplify their job [16, 34, 35, 38]. Witnesses are especially useful for validating optimizations that change the loop structure. Witnesses do not have to be correct, since they are validated by the TV tool. When the compiler provides sufficient information, validation can be done mostly syntactically [17]; there is something of a tradeoff between the amount of changes required in the compiler and the computational and implementation complexity of the TV tool.

Some tools, such as CoVaC [49], Counter [13], DDEC [40], JTFG [9], and trace alignment [6], search for cut points between source and target programs such that some relation between the two programs can be automatically synthesized. When such relations are found, verification can be split into smaller tasks. Moreover, this technique supports some control-flow-changing transformations. TVOC [1, 52] also has heuristics to support transformations that change loop structure.

LLVM-MD [43] and Peggy [42] are TV tools for LLVM that work by rewriting the source program until it is syntactically equal to the source. This process—equality saturation—is

similar to how many first-order theorem provers work (e-matching). egg [46] is a library that implements equality saturation. These tools are limited to proving equivalence.

Coeus [5] implements verification of relational program properties with reinforcement learning. Klebanov et al. [18] proposed a CEGAR-based approach for the verification of program equivalence. Inter-procedural equivalence checking with mutual summaries was proposed in [15, 47]. Compositional Lifter [11] validates binary-to-LLVM IR lifting tools.

A different class of TV tools are ones that are specific to a particular transformation. For example, there are TV tools specific for lazy code motion [44], software pipelining [28, 45], and optimizations for scientific programs [41]. The advantage of being specific is that these tools are simpler than generic ones. Moreover, some of these TV tools are formally verified, which is harder to do for generic tools. Sewell et al. [39] implemented TV for a specific program (seL4 kernel) for its compilation from C to ARM assembly.

While the majority of work so far on TV has focused on equivalence checking, there is one that introduced support for some forms of UB [10]. This work shows that adding support for UB (even if partially) reduces the number of false alarms substantially.

***Formal semantics of LLVM IR.*** There have been several efforts to formalize the semantics of LLVM's IR. Vellvm [51] gives the semantics of parts of the IR with limited support for undefined behavior (UB). The semantics of UB in LLVM IR was further explored in [24]. This work uncovered fundamental issues in LLVM's IR and proposed changes later adopted by LLVM.

The twin memory model [21] is a proposal for a memory model for LLVM that supports UB. A formalization of part of LLVM IR's concurrency instructions was proposed in [4]. K-LLVM [29] is a formalization of LLVM's IR in K with partial support for memory and concurrency.

***Compiler verification.*** An alternative to translation validation is compiler verification, where the compiler/optimizer is verified once and for all. CompCert [27] is a compiler for C that is formalized and verified in Coq. Further work has extended CompCert with verified peephole optimizations [33], verified polyhedral model-based optimizations [8], and a verified SSA-based middle-end optimizer [2].

Cobalt [25] and its successor Rhodium [26] are frameworks to specify and automatically verify peephole optimizations and dataflow analyses. PEC [19] extends this work with support for loop-manipulating optimizations by reusing some of the TVOC's techniques [52].

Alive [30] is an automatic verification tool for LLVM's peephole optimizations. AliveInLean [22] is a reimplementation of Alive that was specified and verified in Lean. Newcomb et al. [36] present an automatic verification tool for soundness and termination of Halide's rewriting system. CORK [31] is an automatic equivalence checker that supports loop optimizations over rational numbers.

***Compiler Fuzzing.*** Fuzzing tools have been very successful in finding bugs in optimizers. Tools like Csmith [48], EMI [20], and SPE [50] have found hundreds of bugs in commercial compilers, including GCC, LLVM, and MSVC. Marcozzi et al. [32] studied the impact of bugs found by fuzzers and verification tools.

## 10 Conclusion

Software development is a fundamentally human process, and there are many opportunities for a large, decentralized group of compiler developers, who primarily coordinate using a mailing list and an English-language specification, to introduce subtle defects into their implementation. To assist the LLVM community in creating a coherent semantics for their IR, and making their toolchain respect it, we have created and deployed Alive2, a tool for bounded translation validation for LLVM IR. Running Alive2 over LLVM's unit test suite has revealed 47 bugs of which 28 have been fixed so far. Moreover, there have been a number of cases where the LLVM IR specification was either vague or defective, and we have worked with the community to fix these. Our goals are to create a formalization of the intended semantics of LLVM IR, to bring the compiler implementation into conformance with these semantics, and to give the LLVM community tools that it can use to prevent deviations from its specification in the future.

## Acknowledgments

The authors thank Alastair Reid, Andrey Rybalchenko, and the anonymous reviewers for feedback on previous versions of this paper. We also thank Pranav Kant and Roman Lebedev for their contributions to Alive2, as well as to the countless people from the LLVM community that have fixed bugs found by Alive2 and that engaged in discussions about IR semantics.

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF-2020R1A2C2011947) and by the Office of Naval Research under Grant No. N00014-17-1-2996.

## References

[1] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. 2005. TVOC: A Translation Validator for Optimizing Compilers. In *CAV*. https://doi.org/10.1007/11513988_29

[2] Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 4 (March 2014). https://doi.org/10.1145/2579080

[3] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In *CC*. https://doi.org/10.1007/978-3-642-37051-9_6

[4] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the Concurrency Semantics of an LLVM Fragment. In *CGO*. https://doi.org/10.1109/CGO.2017.7863732

[5] Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. 2019. Relational Verification Using Reinforcement Learning. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 141 (Oct. 2019). https://doi.org/10.1145/3360567

[6] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *PLDI*. https://doi.org/10.1145/3314221.3314596

[7] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. 2001. *A Simple, Fast Dominance Algorithm.* Technical Report TR-06-33870. Rice University.

[8] Nathanaël Courant and Xavier Leroy. 2021. Verified Code Generation for the Polyhedral Model. In *POPL*. https://doi.org/110.1145/3434321

[9] Manjeet Dahiya and Sorav Bansal. 2017. Black-Box Equivalence Checking Across Compiler Optimizations. In *APLAS*. https://doi.org/10.1007/978-3-319-71237-6_7

[10] Manjeet Dahiya and Sorav Bansal. 2017. Modeling Undefined Behaviour Semantics for Checking Equivalence Across Compiler Optimizations. In *HVC*. https://doi.org/10.1007/978-3-319-70389-3_2

[11] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, and Christopher W. Fletcher. 2020. Scalable Validation of Binary Lifters. In *PLDI*. https://doi.org/10.1145/3385412.3385964

[12] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. https://doi.org/10.1007/978-3-540-78800-3_24

[13] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. 2020. Counterexample-Guided Correlation Algorithm for Translation Validation. In *OOPSLA*. https://doi.org/10.1145/3428289

[14] Paul Havlak. 1997. Nesting of Reducible and Irreducible Loops. *ACM Trans. Program. Lang. Syst.* 19, 4 (July 1997), 557–567. https://doi.org/10.1145/262004.262005

[15] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. 2013. Towards Modularly Comparing Programs Using Automated Theorem Provers. In *CADE*. https://doi.org/10.1007/978-3-642-38574-2_20

[16] Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. 2009. Validation of GCC optimizers through trace generation. *SP&E* 39, 6 (April 2009), 611–639. https://doi.org/10.1002/spe.913

[17] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: Verified Credible Compilation for LLVM. In *PLDI*. https://doi.org/10.1145/3192366.3192377

[18] Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2018. Automating Regression Verification of Pointer Programs by Predicate Abstraction. *Form. Methods Syst. Des.* 52, 3 (June 2018), 229–259. https://doi.org/10.1007/s10703-017-0293-8

[19] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. In *PLDI*. https://doi.org/10.1145/1542476.1542513

[20] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *PLDI*. https://doi.org/10.1145/2594291.2594334

[21] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling High-Level Optimizations and Low-Level Code in LLVM. *Proc. of the ACM on Programming Languages* 2, OOPSLA (Nov. 2018). https://doi.org/10.1145/3276495

[22] Juneyoung Lee, Chung-Kil Hur, and Nuno P. Lopes. 2019. AliveInLean: A Verified LLVM Peephole Optimization Verifier. In *CAV*. https://doi.org/10.1007/978-3-030-25543-5_25

[23] Juneyoung Lee, Dongjoo Kim, Chung-Kil Hur, and Nuno P. Lopes. 2021. An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation. In *CAV*.

[24] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *PLDI*. https://doi.org/10.1145/3062341.3062343

[25] Sorin Lerner, Todd Millstein, and Craig Chambers. 2003. Automatically Proving the Correctness of Compiler Optimizations. In *PLDI*. https://doi.org/10.1145/781131.781156

[26] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. 2005. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. In *POPL*. https://doi.org/10.1145/1040305.1040335

[27] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[28] Raya Leviathan and Amir Pnueli. 2002. Validating software pipelining optimizations. In *CASES*. https://doi.org/10.1145/581630.581676

[29] Liyi Li and Elsa L. Gunter. 2020. K-LLVM: A Relatively Complete Semantics of LLVM IR. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.7

[30] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *PLDI*. https://doi.org/10.1145/2737924.2737965

[31] Nuno P. Lopes and José Monteiro. 2016. Automatic Equivalence Checking of Programs with Uninterpreted Functions and Integer Arithmetic. *Int. J. Softw. Tools Technol. Transf.* 18, 4 (Aug. 2016), 359–374. https://doi.org/10.1007/s10009-015-0366-1

[32] Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. 2019. Compiler Fuzzing: How Much Does It Matter? *Proc. ACM Program. Lang.* 3, OOPSLA (Oct. 2019). https://doi.org/10.1145/3360581

[33] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified Peephole Optimizations for CompCert. In *PLDI*. https://doi.org/10.1145/2908080.2908109

[34] Kedar S. Namjoshi and Lenore D. Zuck. 2013. Witnessing Program Transformations. In *SAS*. https://doi.org/10.1007/978-3-642-38856-9_17

[35] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *PLDI*. https://doi.org/10.1145/349299.349314

[36] Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. 2020. Verifying and Improving Halide's Term Rewriting System with Program Synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 166 (Nov. 2020). https://doi.org/10.1145/3428234

[37] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *TACAS*. https://doi.org/10.1007/BFb0054170

[38] Martin C. Rinard and Darko Marinov. 1999. Credible Compilation with Pointers. In *RTRV*.

[39] Thomas Sewell, Magnus Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *PLDI*. https://doi.org/10.1145/2491956.2462183

[40] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-Driven Equivalence Checking. In *OOPSLA*. https://doi.org/10.1145/2509136.2509509

[41] K.C. Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. 2002. Geometric Model Checking: An Automatic Verification Technique for Loop and Data Reuse Transformations. *ENTCS* 65, 2 (2002). https://doi.org/10.1016/S1571-0661(04)80397-9

[42] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-Based Translation Validator for LLVM. In *CAV*. https://doi.org/10.1007/978-3-642-22110-159

[43] Jean-Baptiste Tristan, Paul Govereau, and J. Gregory Morrisett. 2011. Evaluating value-graph translation validation for LLVM. In *PLDI*. https://doi.org/10.1145/1993316.1993533

[44] Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified Validation of Lazy Code Motion. In *PLDI*. https://doi.org/10.1145/1542476.1542512

[45] Jean-Baptiste Tristan and Xavier Leroy. 2010. A simple, verified validator for software pipelining. In *POPL*. https://doi.org/10.1145/1707801.1706311

[46] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021). https://doi.org/10.1145/3434304

[47] Tim Wood, Sophia Drossopolou, Shuvendu K. Lahiri, and Susan Eisenbach. 2017. Modular Verification of Procedure Equivalence in the Presence of Memory Allocation. In *ESOP*. https://doi.org/10.1007/978-3-662-54434-1_35

[48] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*. https://doi.org/10.1145/1993498.1993532

[49] Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product. In *FM*. https://doi.org/10.1007/978-3-540-68237-0_5

[50] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *PLDI*. https://doi.org/10.1145/3062341.3062379

[51] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *POPL*. https://doi.org/10.1145/2103656.2103709

[52] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. 2005. Translation and Run-Time Validation of Loop Transformations. *Form. Methods Syst. Des.* 27, 3 (Nov. 2005), 335–360. https://doi.org/10.1007/s10703-005-3402-z