Extensible Verified Translation Validation for LLVM

Abstract

Both efficiency and reliability are important for compilers. However, mainstream compilers tend to focus more on efficiency than on reliability. They are complex pieces of software usually written in C/C++ performing various optimizations, thereby containing many bugs. Indeed, recent random testing tools found hundreds of bugs in GCC and LLVM.

To increase the reliability of mainstream compilers without sacrificing their performance, we propose *extensible verified translation validation*. The key ideas are four-fold. First, we perform translation validation. Since we compile first and validate the result later, we do not sacrifice any compilation performance. Second, we modify the compiler itself to generate validation proofs of translations. This way we can always produce correct explicit proofs, or identify compiler bugs. Third, we design an extensible validator that validates the proofs given by the compiler. Our validator is based on relational Hoare logic and can be easily specialized to a particular optimization by simply adding new inference rules. Finally, we can also verify the validator and inference rules in order to reduce the trusted computing base.

We apply our approach to validating 146 micro-optimizations of the `instcombine` pass, the main GVN-PRE algorithm of the `gvn` pass, and register promotion in the `sroa` pass in LLVM 3.7.1. We found three compiler bugs (two fixed, one confirmed) during the development. We ran our validator for SPEC CINT2006 benchmarks and other five projects written in C, with 3.9M lines of code in total. We succeeded at validating 1M translation steps, except for 254 failures, all of which are due to the three compiler bugs we found. We also formally verified correctness of the core part of the validator in the Coq proof assistant.

1. Introduction

Mainstream compilers such as GCC and LLVM are complex pieces of software satisfying various requirements. They are required to generate very efficient code and thus perform many complex optimizations. Also the compilers themselves are required to be very efficient (*i.e.,* to consume less time and memory during compilation) and thus are usually written in C/C++ using sophisticated data structures.

Due to this complexity, it is hard to make mainstream compilers very reliable, which may be seen as one of the most important requirements. Indeed, like other large complex software, current mainstream compilers have many bugs. For example, recent random testing tools such as CSmith [24, 8] and EMI [12] have found hundreds of bugs in GCC and LLVM.

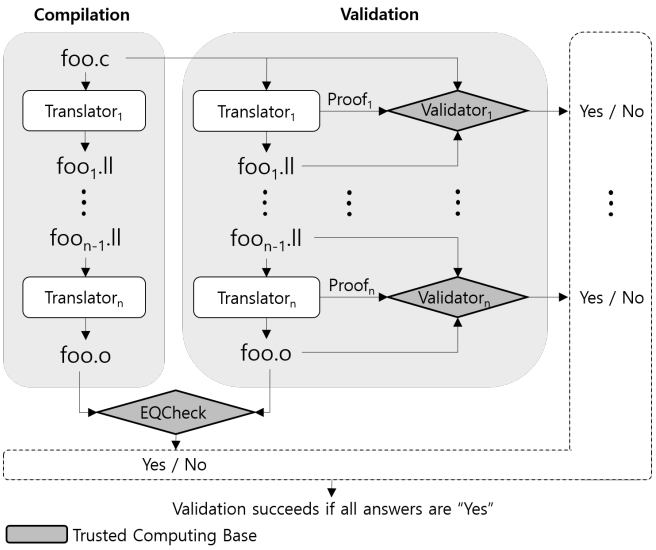


Figure 1. Framework

In order to achieve a high level of reliability without sacrificing performance, we propose a practical approach to compilation verification, which we call *extensible verified translation validation*, and demonstrate its applicability to mainstream compilers. We apply our approach to the global value numbering, the partial redundancy elimination, the register promotion, and 146 micro-optimizations of the instruction combining optimizations in LLVM 3.7.1.

In order to achieve our goal, we design our framework by combining the best ideas from existing approaches (see §1.1) and develop an *extensible relational Hoare-logic (ERHL)* validator to implement the framework (see §1.2).

1.1 Our Framework

We briefly introduce our framework, which is a combination of (verified) translation validation [20, 19, 21, 23, 22, 18] and (foundational) proof carrying code [16, 6] (see §7.1 for comparison).

Problems The main problem with verifying mainstream compilers is simply that they are too large to directly verify. For example, GCC and LLVM consist of million lines of code. Our solution for this problem is to follow the translation

validation approach [20, 19]. The idea is that, instead of verifying the compiler itself, we validate the correctness of compilation results.

Translation Validation with Proofs In our framework, shown in Figure 1, we separate compilation and validation phases. For compilation, depicted in LHS of Figure 1, we use the original compiler to translate, for example, the source code `foo.c` to the target code `foo.o`, which is internally translated through intermediate representations $\texttt{foo}_1\texttt{.ll}, \ldots, \texttt{foo}_{n-1}\texttt{.ll}$. At any time after the compilation, we can conduct post validation, depicted in RHS of Figure 1. For this, we first run the original compiler modified to additionally generate proofs of correctness, $\texttt{Proof}_i$. Then the validators $\texttt{Validator}_i$ check whether $\texttt{foo}_{i-1}\texttt{.ll}$ is correctly translated to $\texttt{foo}_i\texttt{.ll}$ using $\texttt{Proof}_i$.

Finally, we succeed at validation of the original compilation if each validation conducted by $\texttt{Validator}_i$ succeeds and the final result of the original compilation (`foo.o` in LHS) coincides with that of the compilation for validation (`foo.o` in RHS).

Verification of Validators In our framework, the trusted computing base (TCB) includes only the validators $\texttt{Validator}_i$ and the equality checker `EQCheck`. In particular, the proof generation code in the modified compiler is not a part of TCB because any incorrect proof would be invalidated by the validator.

Optionally, we can further reduce the TCB by verifying the correctness of validators: whenever validation of a translation succeeds, the translation is semantics-preserving. Indeed, we implement our validator in the Coq proof assistant and prove correctness of its core using the formal semantics of LLVM IR (Intermediate Representation) from the Vellvm project [26]. In this case, the TCB reduces to the validity of the formal semantics and the correctness of the Coq proof checker.[1]

Advantages The advantages of our framework is four-fold. First, our framework does not sacrifice compilation performance because we completely separate compilation from validation. Second, although the post validation may take long, the validation time can be reduced dramatically by parallellizing the validation jobs because they are completely independent. Third, we can always produce correct explicit proofs of translation, unless there is a compiler bug, since we can get enough information from the compiler. Finally, we can verify the validators to achieve the highest level of reliability, as in fully verified compilers such as CompCert [13]

1.2 Extensible Relational Hoare-Logic Validator

Now the main question remaining is how to generate and validate proofs.

[1] In our experiment, we did not target the frontend translation from `foo.c` to $\texttt{foo}_1\texttt{.ll}$ and the backend translation from $\texttt{foo}_{n-1}\texttt{.ll}$ to `foo.o`.

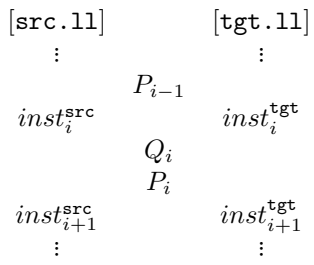
Problems A possible solution would be to follow the verified validation approach [21, 23, 22, 18]. For each translator $\texttt{Translator}_i$, we develop a highly specialized validator $\texttt{Validator}_i$ that fully understands how $\texttt{Translator}_i$ works and validates the correctness of its translation results, when necessary, using some helper information $\texttt{Proof}_i$ given by $\texttt{Translator}_i$.

A problem with this approach is that it is very costly to develop a new validator for each translation pass in mainstream compilers. Typically a mainstream compiler such as GCC and LLVM includes over a hundred major translation passes. Things get even worse if we want to verify the validators: we have to verify over a hundred validators!

Our Approach To solve this problem, we take a different approach. We develop a *single* general validator that can validate correctness proofs given by different translation passes. Here, the general validator is a simple proof checker whereas the compiler provides detailed proofs.

Our general validator, called *extensible relational Hoare-logic (ERHL)* validator, can be applied to translation passes that do not change the control flow graph. For those that change the control flow graph such as loop unrolling or loop unswitching, we need to develop a more general validator, which we leave as future work.

To give a high level idea, we briefly illustrate what constitutes a proof and how it is validated in our ERHL system. For this, suppose we validate the translation from `src.ll` to `tgt.ll` with `Proof`. Also, for simplicity of presentation, suppose the source and target programs consist of a sequence of instructions without any branch.



1. **Instruction Alignment** `Proof` inserts logical no-op instructions to align the instructions of `src.ll` and `tgt.ll` one by one as shown above. This is possible because the translation does not change the control flow.
2. **Invariant Generation** `Proof` provides a relational invariant P_i between program states of `src.ll` and `tgt.ll` after each pair $(inst_i^{\texttt{src}}, inst_i^{\texttt{tgt}})$ of source and target instructions.
3. **Post-Invariant Computation** The validator (i) checks that P_0 is the default invariant that should hold for all initial states; (ii) checks that $inst_i^{\texttt{src}}$ and $inst_i^{\texttt{tgt}}$ produce the same observable events (*e.g.,* calling the same function with the same arguments) under all pairs of program states satisfying P_{i-1}; and (iii) computes a post-invariant

Q_i that should hold for all program states given after executing $inst_i^{\texttt{src}}$ and $inst_i^{\texttt{tgt}}$ under any program states satisfying P_{i-1}. This validator is completely independent from the translation passes and tries to generate as strong post-invariants as possible.

4. **Inference Rule Application** Finally, `Proof` shows the validator how to prove Q_i implies P_i (*i.e.,* any pair of program states satisfying Q_i also satisfies P_i). For this, we pre-install, in the validator, *inference rules* possibly specific to particular translation passes. Then `Proof` provides the validator with a sequence of inference rules to apply in order to infer P_i from Q_i.

We can easily see that the source and target programs produce the same observable events if the validation succeeds.[2] Here the TCB includes the correctness of the validator (mainly, that of the post-invariant computation) and that of the inference rules added. We can optionally verify them to reduce the TCB.

Applicability Our validator is easily extensible: for supporting a new optimization, you just need to install new inference rules possibly specialized for the optimization and verify them. It costs much lower than implementing and verifying a specialized validator for the optimization.

We believe our approach applies to most of the optimizations that do not change control flow graph, which comprises 66.7% of the `-O2` optimization passes of LLVM 3.7.1. This is because the majority of compiler optimizations in LLVM rely on relatively simple (flow-sensitive) analysis, which we can handle in a relational Hoare logic.

1.3 Result

We developed a prototype ERHL validator for the LLVM compiler, and successfully validated compilations of 146 micro-optimizations of the `instcombine` pass, the main GVN-PRE algorithm of the `gvn` pass, and register promotion in the `sroa` pass for SPEC CINT2006 C benchmarks [5] and other five C projects, with 3.9M lines of code in total. In the benchmark we successfully validated 1,191,063 translations, except for 254 failures that are due to the three compiler bugs we found during this project, and 197,652 (16.6%) that are currently not supported in the validator, for example, because they contain instructions that are not formalized yet. We also prove correctness of the core part of the validator in Coq, though leaving admits on minor details.

2. Overview

In this section, we present more details of how our validation system works using the `add-assoc` optimization as a motivating example, which is a part of the `instcombine` pass of the LLVM compiler.

[2] Technically, we show behavioral refinement instead of equivalence. For simplicity of presentation, we use behavioral equivalence for now.

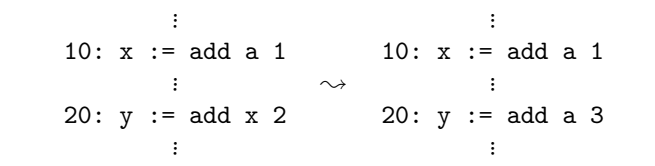


Figure 2. An instance of the `add-assoc` optimization

Algorithm 1 AddAssoc (P:program)

```
1:  for y in Reg(P) do
2:    match FindDef(P, y) with
3:    | Some (l2, add (reg x) (const C2)) ⇒
4:      match FindDef(P, x) with
5:      | Some (l1, add (reg a) (const C1)) ⇒
          Propagate(src, x = add a C1, l1, l2)
          Infrule(src, assoc(add, x, y, a, C1, C2), l2)
          Infrule(reduce_maydiff(y), l2)
6:        C = Simplify(add, C1, C2)
7:        ReplaceAt(P, l2, y = add (reg a) (const C))
8:      | _ ⇒ () end match
9:    | _ ⇒ () end match
10: end for
```

2.1 The `add-assoc` Optimization

Figure 2 shows an example of the `add-assoc` optimization. This gives the opportunity to remove the instruction `10: x := add a 1` later, if `x` is not used anymore. Correctness of this translation is quite clear intuitively. Thanks to the SSA (Static Single Assignment) property [9][3], there are no intervening writes to `a` and `x` between line 10 and 20, and thus at line 20, $\texttt{x} = \texttt{a} + 1$ holds, from which $\texttt{y} = \texttt{x} + 2 = (\texttt{a} + 1) + 2 = \texttt{a} + 3$ follows.

We give a simplified version of the compiler code for `add-assoc`, presented in a functional style for presentation purpose, in Algorithm 1. The code in the dashed box is what we add to generate proofs for validation, which we will discuss later. The algorithm proceeds as follows:

Lines 1-3 For a source program P, we find an instruction of the form l_2`: y := add x` C_2 with C_2 constant. Note that $\text{Reg}(P)$ is the set of all registers defined in P and $\text{FindDef}(P, y)$ finds the instruction defining the register y, which is unique thanks to the SSA property. For *e.g.,* in Figure 2, we can find `20: y := add x 2`.

Lines 4-5 We check if x is defined by an instruction of the form l_1`: x := add a` C_1 with C_1 constant. For *e.g.,* in Figure 2, the register `x` is defined by `10: x := add a 1`.

Lines 6-7 If that is the case, we compute the constant C `= add` C_1 C_2 and replace the instruction at l_2 with `y := add a` C. For *e.g.,* in Figure 2, `20: y := add x 2` is replaced by `20: y := add a 3`.

[3] The SSA property is that for every register, there is exactly one instruction defining it (*i.e.,* assigning a value to it).

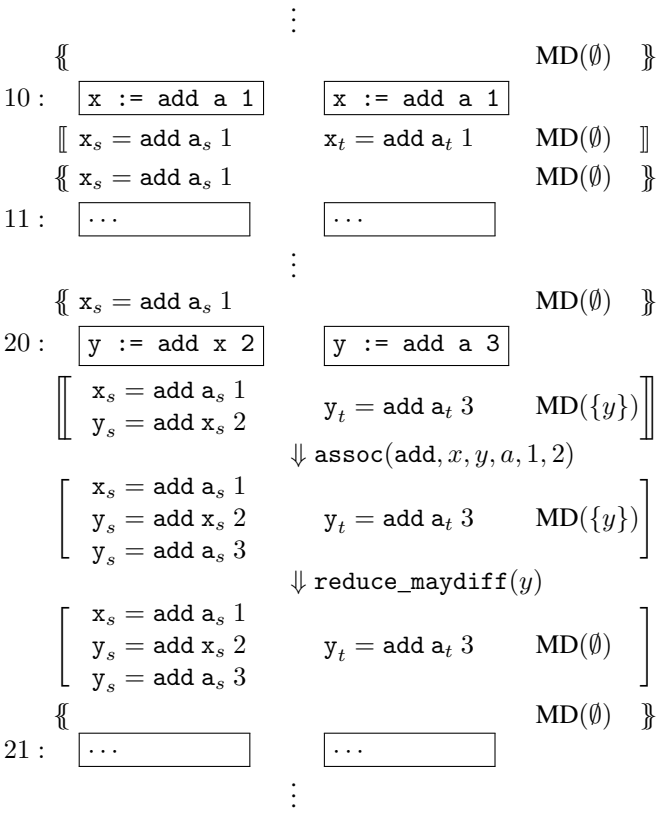


Figure 3. Validation of an `add-assoc` proof in ERHL

2.2 Generation and Validation of Translation Proof

Now we generate and validate proofs for `add-assoc` translations in ERHL, as outlined in the introduction. To help understanding, Figure 3 shows the proof and its validation for the example translation of Figure 2. First, we do not need to insert any logical no-ops because the source and target programs are already well aligned.

2.2.1 Invariant Generation

Invariant Format Before we proceed, we show the format of invariants in ERHL. An ERHL invariant is a triple (S, T, M), where S is an invariant that should hold for the source state; T is an invariant for the target state; and M is an invariant relating the source and target states.

First, every ERHL invariant implicitly impose that the source and target memories should be equivalent. More precisely, we use the CompCert-style memory injection relation [14] (see §3.5 for more details).

Second, the source and target invariants, S and T, are sets of propositions that should hold for the source and target states, each of which can contain various forms of predicates. Though we only use the equality predicate for `add-assoc`, we will see more predicates for other optimizations in §3. As examples, $\{\, \mathtt{x}_s = \mathtt{add}\ \mathtt{a}_s\ 1,\ \mathtt{y}_s = \mathtt{add}\ \mathtt{x}_s\ 2 \,\}$ is a source invariant and $\{\, \mathtt{y}_t = \mathtt{add}\ \mathtt{a}_t\ 3 \,\}$ is a target invariant. Here and henceforth, x_s and x_t represent the value of the register x in the source and target states, respectively.

Finally, the invariant M is a set of registers, called *maydiff set*, that may contain different values in the source and target states. In other words, all the registers not in M should have the same value in the source and the target states, which we denote by $\mathrm{MD}(M)$:

$$\mathrm{MD}(M) \iff \forall x \notin M.\ x_s = x_t\ .$$

A main advantage of ERHL invariants is that we can use the standard algorithm of Hoare logic to compute post invariants because ERHL invariants are mainly unary (only maydiff sets are relational). Though mainly unary, ERHL invariants can encode more general forms of relational invariants (see §3.3), which we believe are sufficiently expressive to handle standard compiler optimizations (preserving the control flow).

Invariant Generation As shown in Figure 3, we generate an appropriate ERHL invariant at each line (see those in {{ }}). In ERHL, the default invariant is $(\emptyset, \emptyset, \emptyset)$, which denotes that the source and target memories are equivalent and the source and target registers contain the same values.

To prove correctness of `add-assoc`, we need to know $x_s = \mathtt{add}\ a_s\ C_1$ at the point of defining y, and thus we add $x_s = \mathtt{add}\ a_s\ C_1$ to the source invariants between line l_1, where x is defined, and line l_2, where y is defined. The code $\mathrm{Propagate}(\mathtt{src}, x = \mathtt{add}\ a\ C_1, l_1, l_2)$ in the dashed box of Algorithm 1 generates such additional invariants. More precisely, in order to reduce the proof size, the compiler just outputs the command $\mathrm{Propagate}(\mathtt{src}, x = \mathtt{add}\ a\ C_1, l_1, l_2)$, from which a preprocessor in the validator, called *invariant propagator*, generates the full invariants. Note that the invariant propagator is not a part of the TCB.

2.2.2 Post-Invariant Computation

As outlined in the introduction, the ERHL validator proceeds in three steps. First, it checks whether the initial invariant is the default invariant $(\emptyset, \emptyset, \emptyset)$, which is the case in Figure 3. Second, it checks whether the same observable events are produced at each line. It is the case in Figure 3 because (i) at line 20, no observable events are produced; and (ii) at the other lines, the source and target instructions are identical and the maydiff sets are empty implying that the source and target states are equivalent.

Finally, at each line in Figure 3, the validator computes a post-invariant, that in [[ ]] just after the line, from the pre-invariant, that in {{ }} just before the line. The post-invariant computation algorithm is simple. It basically (i) removes from the source and target invariants the propositions that may be invalidated by the instructions executed; (ii) adds to the source and target invariants new propositions induced from the instructions executed; and (iii) adds to the maydiff set the registers just defined if the instructions executed are not identical, or their arguments are in the maydiff set. In

Figure 3, for example, at line 10 we add $\texttt{x}_s = \texttt{add a}_s\ 1$ and $\texttt{x}_t = \texttt{add a}_t\ 1$; and at line 20 we add $\texttt{y}_s = \texttt{add x}_s\ 2$, $\texttt{y}_t = \texttt{add a}_t\ 3$ and MD($\{y\}$).

2.2.3 Inference Rule Application

We finally infer, at each line, the invariant in $\{\!\{\ \}\!\}$ given by the compiler, from that in $[\![\ ]\!]$ computed by the validator.

Except line 20, the inference is done automatically by the validator because it is trivial. In particular, since there are no intervening writes to `a` and `x` between line 10 and 20 thanks to the SSA property, the invariant $\texttt{x}_s = \texttt{add a}_s\ 1$ is never removed by the post-invariant computation algorithm, so that the invariant trivially holds at the following line.

At line 20, the inference is guided by the proof from the compiler. First, the code Infrule(`src`, `assoc`(`add`, x, y, a, C_1, C_2), l_2) in the dashed box of Algorithm 1 outputs a command that applies the following pre-installed `assoc` rule with the given arguments at line l_2.

$$\frac{(\texttt{assoc}(\texttt{add}, x, y, a, C_1, C_2)) \qquad x = \texttt{add}\ a\ C_1 \qquad y = \texttt{add}\ x\ C_2 \qquad C = C_1 + C_2}{y = \texttt{add}\ a\ C}$$

In Figure 3, we have $\texttt{y}_s = \texttt{add a}_s\ 3$ by this inference rule.

Second, the code Infrule(`reduce_maydiff`(y), l_2) in Algorithm 1 outputs a command that applies the following pre-installed `reduce_maydiff` rule with the argument y at line l_2:

$$\frac{(\texttt{reduce\_maydiff}(y)) \qquad y_s = e_s \qquad y_t = e_t \qquad \text{no reg in } e \text{ is in maydiff}}{y_s = y_t}$$

where and henceforth e_s and e_t denote the expression e with every register x replaced by x_s and x_t, respectively. This rule removes y from the maydiff set by inferring the equality $y_s = y_t$. In Figure 3, the register `y` is removed from the maydiff set by this rule because we have $\texttt{y}_s = \texttt{add a}_s\ 3$ and $\texttt{y}_t = \texttt{add a}_t\ 3$ with the register `a` not in the maydiff set.

Finally, the last step of inference at line 20 between the invariant in $[\,]$ and that in $\{\!\{\ \}\!\}$ is trivial and performed automatically by the validator.

3. Key Ideas behind ERHL

We present key ideas for validating various translations using examples taken from our experiment. For simplicity of presentation, we will henceforth use C-like syntax instead of the LLVM syntax (*e.g.,* $x := y + z$ for $x := \texttt{add}\ y\ z$).

3.1 Aligning Instructions with Logical No-ops

There are optimizations that break the one-to-one correspondence between the source and target instructions such as dead code elimination (DCE). For such optimizations, we align the instructions by inserting the *logical no-op* instruction `lnop`.

The instruction `lnop` is logical because it is absent from the real code and used only for validation purpose. In validation, it is interpreted as doing nothing (*i.e.,* no-op). For example, in the following DCE translation, we can insert `lnop` in the target to align the source and target instructions.

```
...                    ...
y := x + 42;    ⇝      (lnop;)  // y not used
...                    ...
```

3.2 Validating Memory Optimizations

We show how ERHL handles non-alias information between pointers and private memory locations in order to validate memory optimizations.

Simple Alias Analysis Optimizations about memory operations are hard because pointers may be *aliased*, *i.e.,* pointer values of different registers may point to the same location. For example, the `load-load` optimization in the `instcombine` pass of LLVM performs the following translation because `p` and `q` are not aliased.

```
10: p := alloc(4);        p := alloc(4);
11: *p := 42;            *p := 42;
12: *q := 37;      ⇝     *q := 37;
13: r := *p;              (lnop;)
14: foo(r);               foo(42);
```

To reason about such translations, we introduce the *noalias* predicate: $p \perp q$ denotes that the values in the registers p and q are not aliased (*i.e.,* pointing to disjoint memory blocks) if they are pointers. We also adapt the post-invariant generator: all propositions about $*p$ are preserved by a store instruction to $*q$ if we have $p \perp q$ in the pre-invariant; otherwise all dropped. Hence the above translation can be validated in ERHL, if we can derive $\texttt{p}_s \perp \texttt{q}_s$ before `*q := 37`.

To derive and reason about noalias predicates, we introduce a stronger predicate, Unique(p). It asserts that the value in the register p is not aliased with any values in the other registers or in the memory. Accordingly, the post-invariant generator introduces Unique(p) after $p :=$ `alloc(..)` and drops it after (almost) all other instructions. Also we install the following inference rules:

$$\frac{(\text{UNIQUE-NOALIAS}) \qquad \text{Unique}(p) \qquad p, q \text{ different registers}}{p \perp q}$$

Using such predicates, we can validate the above translation as follows. We have: after line 10, Unique($\texttt{p}_s$) as a post-invariant and $\texttt{p}_s \perp \texttt{q}_s$ by (UNIQUE-NOALIAS); after line 11, $*\texttt{p}_s = 42$ as a post-invariant and $\texttt{p}_s \perp \texttt{q}_s$ preserved; and after line 12, $*\texttt{p}_s = 42$ preserved thanks to $\texttt{p}_s \perp \texttt{q}_s$. Thus after line 13, we can infer that $\texttt{r}_s = 42$ using $*\texttt{p}_s = 42$, from which the validator can check that `foo` is invoked with 42 in both source and target.

Note that the LLVM infrastructure provides an alias analysis module that performs more complex analysis. Supporting it in ERHL is future work (see §7.2).

Private Memory Another challenge in validating memory optimizations is to handle unknown code. For example, consider the following register promotion translation performed by the `sroa` pass[4], where the address in `p` is used locally.

```
10:  p := alloc(4);          (lnop;)
11: *p := 42;                (lnop;)
12:  bar();          ⇝       bar();
13:  r := *p;                (lnop;)
14:  foo(r);                 foo(42);
```

The `sroa` pass performs this translation even though the function `bar` is unknown at compile time because the address in `p` is newly allocated and not escaped to external functions and thus the function `bar` has no way to access the address.

To make this argument formal, we introduce the *private memory* predicate: $\mathrm{Private}(p_s)$ denotes that the register p contains a pointer value pointing to a *private* block in the following sense. First, in our semantic model for validation, we have the notion of *equivalent* blocks between the source and target memories, which is technically a CompCert-style memory injection (see §3.5 for more details). Then, a block in the source memory is *public* if it has a corresponding equivalent block in the target memory; otherwise *private*. Note that the values of the registers not in the maydiff set and the values stored in the public blocks are also public.

Regarding the predicate, the post-invariant generator is adapted in three ways. First, $\mathrm{Private}(p_s)$ is introduced after the line with $p :=$ `alloc(..)` in source and `lnop` in target. Second, $\mathrm{Private}(p_s)$ is preserved by every instruction since the validator checks that only public values are passed to external functions and stored in public blocks.[5] Third, all propositions about $*p$ are preserved by a function call if we have $\mathrm{Private}(p_s)$ in the pre-invariant; otherwise all dropped.

Now we can validate the above translation as follows. We have: after line 10, $\mathrm{Private}(\mathtt{p}_s)$ as a post-invariant; after line 11, $*\mathtt{p}_s = 42$ as a post-invariant and $\mathrm{Private}(\mathtt{p}_s)$ preserved; and after line 12, $*\mathtt{p}_s = 42$ preserved thanks to $\mathrm{Private}(\mathtt{p}_s)$. Thus after line 13, we can infer that $\mathtt{r}_s = 42$ using $*\mathtt{p}_s = 42$, from which the validator can check that `foo` is invoked with 42 in both source and target.

Note that in the presence of both function calls and store instructions, we need to use all three predicates: $\mathrm{Unique}(p)$, $p \perp q$ and $\mathrm{Private}(p)$.

3.3 Encoding Relational Predicates using Ghosts

We show how to encode a more general form of relational invariant, $e_s = e'_t$, which is often needed, for instance, to handle loops.

The idea is to use *ghost registers* to relate the source and target expressions. Specifically, we can encode $e_s = e'_t$ as follows: for a fresh ghost register $\hat{\mathtt{g}}$,

$$e_s = \hat{\mathtt{g}}_s \ \wedge\ \hat{\mathtt{g}}_t = e'_t \ \wedge\ \hat{\mathtt{g}}_s = \hat{\mathtt{g}}_t \ .$$

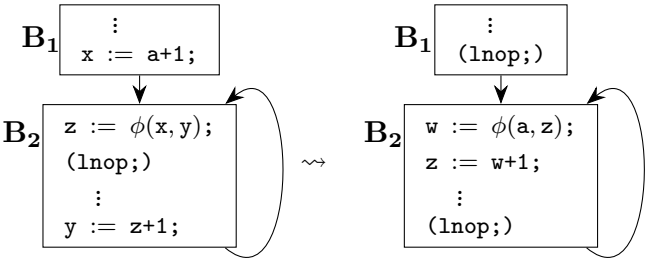
Here, $e_s = \hat{\mathtt{g}}_s$ is a source invariant, $\hat{\mathtt{g}}_t = e'_t$ is a target invariant and $\hat{\mathtt{g}}_s = \hat{\mathtt{g}}_t$ denotes that $\hat{\mathtt{g}}$ is not in the maydiff set.

Ghost registers are *logical* registers that do not exist in the physical program state. Thus they are existentially quantified in the semantics of ERHL invariants. Specifically, a pair of source and target states (S, T) satisfies an ERHL invariant P, if there *exists* a pair of source and target ghost register states $(\hat{R}_s, \hat{R}_t)$ such that the pair of S extended with $\hat{R}_s$ and T extended with $\hat{R}_t$ satisfy P.

3.4 Handling Phinodes and Loops

We show how we handle phinodes and loops using ghost variables. Handling loops is more challenging because a register may be updated multiple times in the presence of loop even with the SSA property.

We illustrate the idea using an example. Consider the following translation performed by the `fold-phi` optimization in the `instcombine` pass of LLVM.



In the source program, the block $\mathbf{B_1}$ flows into $\mathbf{B_2}$, and the block $\mathbf{B_2}$ makes a self loop. The *phinode* $\mathtt{z} := \phi(\mathtt{x}, \mathtt{y})$ is a construct of SSA, whose semantics is that the register `z` is assigned the value of `x` when control coming from $\mathbf{B_1}$; and the value of `y` when control coming from $\mathbf{B_2}$. Thus at each iteration of the loop the value of `z` gets incremented by 1.

This translation can be validated as follows. We first assert: $\mathrm{MD}(\{\mathtt{x}, \mathtt{y}, \mathtt{w}\})$ throughout $\mathbf{B_1}$ and $\mathbf{B_2}$ (because `x`, `y` are eliminated and `w` is introduced in the target); $\mathtt{x}_s = \mathtt{a}_s + 1$ at the end of $\mathbf{B_1}$; and $\mathtt{y}_s = \mathtt{z}_s + 1$ at the end of $\mathbf{B_2}$. Then, after the phinode in $\mathbf{B_2}$, we assert $\mathrm{MD}(\{\mathtt{x}, \mathtt{y}, \mathtt{w}, \mathtt{z}\})$ and $\mathtt{z}_s = \mathtt{w}_t + 1$, which is encoded as $\mathtt{z}_s = \hat{\mathtt{t}}_s + 1 \wedge \hat{\mathtt{t}}_t = \mathtt{w}_t$ using a fresh ghost register $\hat{\mathtt{t}}$.

Now we can validate these asserted invariants. There are two non-trivial steps.

First step The invariant $\mathtt{z}_s = \hat{\mathtt{t}}_s + 1 \wedge \hat{\mathtt{t}}_t = \mathtt{w}_t$ after the phinode is validated independently for each incoming control flow.

When control coming from $\mathbf{B_1}$, before the phinode, the invariant $\mathtt{a}_s = \hat{\mathtt{t}}_s \wedge \hat{\mathtt{t}}_t = \mathtt{a}_t$ is inferred by the following rule because `a` is not in the maydiff set.

$$\frac{\hat{t} \text{ is fresh} \qquad \text{no reg in } e \text{ is in maydiff}}{e_s = \hat{t}_s \ \wedge\ \hat{t}_t = e_t} \ (\textsc{intro-ghost})$$

[4] The `sroa` pass performs register promotion and replaces the `mem2reg` pass.

[5] However it is possible to install an inference rule that eliminates $\mathrm{Private}(p_s)$ turning a private block into a public one under certain conditions because it is allowed in our semantic model.

This rule is sound because the ghost variable $\hat{t}$ is not used elsewhere (*i.e.,* fresh) and existentially quantified. Then $\mathtt{x}_s = \hat{\mathtt{t}}_s + 1$ is inferred from $\mathtt{x}_s = \mathtt{a}_s + 1$ from $\mathbf{B_1}$ and $\mathtt{a}_s = \hat{\mathtt{t}}_s$ by the following rule.

$$\frac{x = e \qquad y = z}{x = e[y \mapsto z]} \text{(RENAME)}$$

After the phinode, $\mathtt{z}_s = \hat{\mathtt{t}}_s + 1$ is inferred from the post-invariant $\mathtt{z}_s = \mathtt{x}_s$ and the preserved invariant $\mathtt{x}_s = \hat{\mathtt{t}}_s + 1$ by transitivity. Also $\hat{\mathtt{t}}_t = \mathtt{w}_t$ is inferred from the preserved invariant $\hat{\mathtt{t}}_t = \mathtt{a}_t$ and the post-invariant $\mathtt{a}_t = \mathtt{w}_t$ by transitivity.

When control coming from $\mathbf{B_2}$, before the phinode, the invariant $\mathtt{z}_s = \hat{\mathtt{t}}_s \wedge \hat{\mathtt{t}}_t = \mathtt{z}_t$ is inferred by (INTRO-GHOST). Then $\mathtt{y}_s = \hat{\mathtt{t}}_s + 1$ is inferred from $\mathtt{y}_s = \mathtt{z}_s + 1$ from $\mathbf{B_2}$ and $\mathtt{z}_s = \hat{\mathtt{t}}_s$ by (RENAME). After the phinode, $\mathtt{z}_s = \hat{\mathtt{t}}_s + 1$ is inferred from the post-invariant $\mathtt{z}_s = \mathtt{y}_s$ and the preserved invariant $\mathtt{y}_s = \hat{\mathtt{t}}_s + 1$ by transitivity. Also $\hat{\mathtt{t}}_t = \mathtt{w}_t$ is inferred from the preserved invariant $\hat{\mathtt{t}}_t = \mathtt{z}_t$ and the post-invariant $\mathtt{z}_t = \mathtt{w}_t$ by transitivity.

Second step Now the invariant $\mathtt{z}_s = \mathtt{z}_t$ (*i.e.,* removing z from the maydiff set) is validated after the second line of $\mathbf{B_2}$. After `z := w+1`, the invariant $\mathtt{z}_t = \hat{\mathtt{t}}_t + 1$ is inferred from the post-invariant $\mathtt{z}_t = \mathtt{w}_t + 1$ and the preserved pre-invariant $\hat{\mathtt{t}}_t = \mathtt{w}_t$ by (RENAME). Since the pre-invariant $\mathtt{z}_s = \hat{\mathtt{t}}_s + 1$ is also preserved, we have $\mathtt{z}_s = \mathtt{z}_t$ by `reduce_maydiff(z)`.

As we have seen, reasoning in the presence of phinodes is somewhat involved. In order to reduce manual effort, we have equipped the validator with an engine that automatically apply (up to some level) relevant inference rules regarding phinodes. Note that the inference rules used by the engine are parts of TCB, but the automation algorithm itself is not.

3.5 Generalizing Equality

Technically, the correctness of a translation is not equivalence but refinement between the behaviors of the source and target programs. More specifically, the behaviors of the target program need to be a subset of those of the source. Since compiler optimizations indeed exploit this definition, we also reflect it in ERHL by weakening the notion of equality.

For this, we basically follow the ideas from CompCert [13]. We use the notions of *lessdef* and *memory-injection*.

Lessdef The LLVM IR has the notion of *poison* value, which is equivalent to the *undef* value of CompCert. The poison value comes as an outcome of erroneous operation such as integer addition overflow. Since compilers have freedom to translate any erroneous behavior into an arbitrary behavior, the poison value can be translated into an arbitrary value.

Such translations may break the equality relationship between the source and target values. For example, consider the following translation:

```
x := a - 1;        x := a - 1;
y := a - x;   ⇝    y := a - x;
z := y + 1;        z := 1 + 1;
```

The register y is replaced by 1 because $\mathtt{y}_s = \mathtt{a}_s - \mathtt{x}_s = \mathtt{a}_s - (\mathtt{a}_s - 1) = 1$ for any integer value of $\mathtt{a}_s$. However, if $\mathtt{a}_s$ is `poison`, the equality $\mathtt{y}_s = 1$ does not hold because arithmetic operations over `poison` unconditionally yield `poison` and thus $\mathtt{a}_s - (\mathtt{a}_s - 1) = \mathtt{poison}$. As a result, we do not have $\mathtt{z}_s = \mathtt{z}_t$ in this valid translation because $\mathtt{z}_s = \mathtt{poison} \neq 2 = \mathtt{z}_t$ when $\mathtt{a}_s$ is `poison`.

In order to properly capture the relationship between source and target values, we use the CompCert-style *lessdef* relation [14] instead of the equality. Specifically, a is *less defined* than b, denoted $a \sqsupseteq b$, if a is `poison` or equals to b. For example, we have $\mathtt{y}_s = \mathtt{a}_s - \mathtt{x}_s = \mathtt{a}_s - (\mathtt{a}_s - 1) \sqsupseteq 1$ and $\mathtt{z}_s = \mathtt{y}_s + 1 \sqsupseteq 1 + 1 = \mathtt{z}_t$, which justifies the above translation.

Memory Injection The memory blocks in source and target may not have one-to-one correspondence since optimizations may remove, add or merge memory allocations (*e.g.,* register promotion and allocation). Hence the corresponding pointer values in source and target may not have the same address.

In order to capture this relationship between pointer values in source and target, we also use the CompCert-style *memory injection* relation [14] instead of the equality. The idea is two-fold. First, we use logical notions of memory block and address in the operational semantics of LLVM IR. Then, in the simulation relation used to prove behavioral refinement between source and target programs, we keep a relation satisfying certain properties, called memory injection, between the logical memory blocks in source and target to capture the notion of *equivalent* blocks.

Though we use lessdef with memory injection in all our work, we use the equality instead in the paper for simplicity of presentation. However, all ideas presented in the paper naturally generalize to the setting with lessdef and memory injection. For example, after `x := a + b`, the post-invariant generator introduces both $\mathtt{x} \sqsupseteq \mathtt{a} + \mathtt{b}$ and $\mathtt{a} + \mathtt{b} \sqsupseteq \mathtt{x}$.

4. Validation of LLVM Optimizations

We show how we combine the key ideas presented in the previous section to validate translations performed by three LLVM passes: instruction combining (`instcombine`), global value numbering with partial redundancy elimination (`gvn`), and register promotion (`sroa`).

4.1 Instruction Combining

The `instcombine` pass in LLVM is a collection of many micro-optimizations that detects inefficient patterns of instructions and transforms them into a cheaper form. It is one of the biggest passes in LLVM, comprising 23,000 lines of

C++ code with more than 1000 micro-optimizations [15]. Among those micro-optimizations, we extend and apply our validator to selected 146 micro-optimizations. We select at least one micro-optimization from each of the 10 files in the `instcombine` pass.[6]

The selected 146 micro-optimizations are not just very simple "peephole" optimizations. They include non-trivial optimizations: those across phinodes and loops such as `fold-phi` and memory optimizations such as `load-load`, `load-store`, and `dead-store-elim`. Indeed all the ideas of §3 are used as follows. Out of the 146 micro-optimizations:

- 30 use the logical no-op instruction (§3.1);
- 3 use the noalias predicate (§3.2);
- 4 use relational predicates with ghost registers (§3.3,§3.4);
- 5 materialize `poison` into a concrete value (§3.5).

In the supplementary material [1, §C], we present the full list of the 146 micro-optimizations with the techniques used by them.

4.2 Global Value Numbering with Partial Redundancy Elimination

Global Value Numbering The `gvn` pass in LLVM performs global value numbering (GVN), which eliminates redundant instructions whose outcome is statically known equivalent to that of a previous instruction. The GVN algorithm identifies redundant instructions by classifying expressions into equivalence classes of those yielding the same outcome.

As an example, consider the following translation:

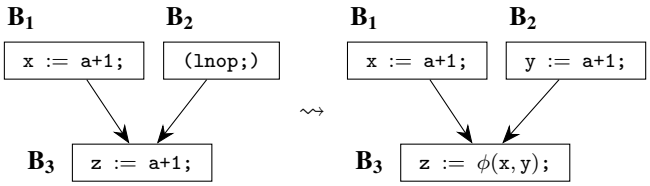
```
x := a + 1;          x := a + 1;
y := a + 1;          (lnop;) // y replaced by x
z := y + 2;   ⇝      z := x + 2;
v := x + 2;          (lnop;) // v replaced by z
w := z + v;          w := z + z;
```

Here, `x` and `y` are classified as results of equivalent expressions and thus `y` is removed and replaced by `x`. Subsequently `z` and `v` are classified as equivalent and thus `v` is removed and replaced by `z`.

The validation of GVN is similar to that of simple arithmetic optimizations like `add-assoc`, but it is required to generate proofs in a recursive fashion, due to the recursive nature of GVN's value classification algorithm.

Partial Redundancy Elimination The `gvn` pass also performs *partial redundancy elimination* (PRE), which eliminates partial redundancy using the classification result of GVN.

As an example, consider the following translation:



Here, `z` in $\mathbf{B_3}$ is partially redundant because it is redundant only when the preceding block is $\mathbf{B_1}$. PRE removes such partial redundancy as follows. First, it inserts `y := a+1` in $\mathbf{B_2}$ to make z fully redundant and then removes the redundancy by transforming `z := a+1` into $\phi(\mathtt{x}, \mathtt{y})$.

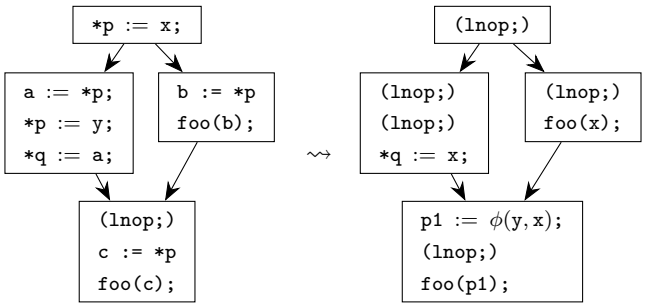
For validation of PRE, we additionally use the idea of §3.3 and §3.4 to perform validation across phinodes and loops. Note that PRE is performed even across loops.

In our experiment, we fully cover the main algorithm, GVN and PRE, of the `gvn` pass, which consists of 1,496 lines of C++ code. However, the `gvn` pass also performs several micro-optimizations that help the main GVN-PRE algorithm, which we do not currently handle.

4.3 Register Promotion

The register promotion optimization optimizes loads and stores to memory locations into reads and writes to registers, provided that the locations are used only locally (*i.e.,* their addresses are not escaped to external functions nor to the memory). Furthermore, it is this register promotion that essentially performs the SSA transformation, since stores to the same location are allowed statically multiple times whereas writes to a register is allowed statically once. This algorithm is implemented in a single file consisting of 989 lines of C++ code, which is invoked by the `sroa` pass (previously, invoked by the `mem2reg` pass).

As an example, consider the following translation in which the location `p` is promoted to existing registers `x`, `y` and a fresh register `p1`:



To validate this translation, most of the ideas in §3 are used. First, the logical no-op instruction (§3.2) are properly inserted. Second, the noalias and private memory predicates (§3.2) are used for preserving invariants about $*\mathtt{p}_s$ after the store instruction `*q := a` and the function calls `foo(b)` and `foo(c)`. Finally, ghost registers (§3.3,§3.4) are used to express relational predicates such as $*\mathtt{p}_s = \mathtt{p1}_t$ after the phinode `p1 := φ(y,x)`. Such phinodes are inserted in the

[6] We omitted the files on vectors, function calls, and bit operations.

presence of branches and loops to keep the target program in SSA form.

Note that the implementation of register promotion in LLVM temporarily breaks the SSA and semantics preservation properties, which makes step-by-step verification hard [25]. In order to sidestep this problem, we validate the whole steps of a register promotion translation at once. More specifically, we accumulate all proofs generated during the whole translation and merge them as a single proof, which we pass to the validator.

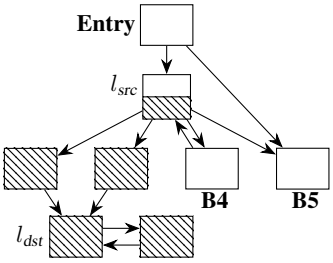
5. Validator Components

In this section, we present further details of the components of our ERHL system.

5.1 Invariant Propagator

As discussed in §2.2.1, the invariant propagator generates full invariants from commands given by the compiler. For example, the command Propagate($\texttt{src}, \texttt{x} = \texttt{add y } 42, l_{src}, l_{dst}$) requires to add the proposition $\texttt{x} = \texttt{add y } 42$, say P shortly, to the source invariants between l_{src} and l_{dst}, where the notion of program point (*e.g.,* l_{src}, l_{dst}) is generalized from a line number to a block name with a line number in it.

More specifically, the proposition P should be added at every program point appearing in a path from l_{src} to l_{dst} that does not visit l_{src} but may visit l_{dst} in-between. Since l_{src} is the source of the proposition P so that we can get P as a post-condition every time we visit l_{src}, there is no need to propagate P along a path from l_{src} to l_{src}. For example, consider the following program.



The marked area between l_{src} and l_{dst} is where we should propagate the proposition P.

Thanks to the SSA property [9], we can efficiently calculate the program points between l_{src} and l_{dst}. First, we can assume that l_{src} *dominates* l_{dst} (*i.e.,* one should have visited l_{src} to reach l_{dst} from the entry point), since the proposition P created at l_{src} should hold at l_{dst}. Then a program point l is on a path from l_{src} to l_{dst} that does not visit l_{src} in-between if and only if (i) l_{src} dominates l and (ii) l_{dst} is reachable from l without visiting l_{src}. We efficiently check the first condition using the dominator tree [9] and the second condition by a backward BFS search from l_{dst}.

In the above example, any program point in the marked area is such that l_{src} dominates it and l_{dst} is reachable from it without visiting l_{src}. On the other hand, l_{dst} is not reachable from the block B_4 without visiting l_{src}, and l_{src} does not dominate the block B_5.

Note that the invariant propagator is not a part of TCB: validation may fail but cannot succeed incorrectly due to bugs in the propagator.

5.2 Post-Invariant Generator

The post-invariant generator, $\texttt{postinv} : \mathit{Instr} \times \mathit{Instr} \times \mathit{Inv} \to \texttt{option } \mathit{Inv}$, computes a post-invariant of the given pre-invariant after the given source and target instructions. As illustrated in §2.2.2, we define it as follows:

$$
\begin{aligned}
&\texttt{postinv}(i_s, i_t, P) = \\
&\quad \texttt{if check}(i_s, i_t, P) \\
&\quad \texttt{then Some auto}(\texttt{add}(i_s, i_t, \texttt{remove}(i_s, i_t, P))) \\
&\quad \texttt{else None}\ .
\end{aligned}
$$

The function $\texttt{check}(i_s, i_t, P)$ checks whether the instructions i_s and i_t produce the same observable events under the program states satisfying the pre-invariant P. Specifically, it checks whether i_s and i_t call the same function with equivalent arguments, if one of them is a function call; and whether i_s and i_t store equivalent values to equivalent memory locations including global variables, if one of them is a store to a public location.

The function `check` checks the equivalence of a register x in source and y in target by checking whether there is a (possibly ghost) register z such that $x_s = z_s \wedge y_t = z_t$ and z is not in the maydiff set. This is enough because we can always make provably equivalent registers in source and target to factor through a fresh ghost register by transitivity and (INTRO-GHOST).

The function $\texttt{remove}(i_s, i_t, P)$ computes a post-invariant Q by removing from the pre-invariant P those propositions that are invalidated by executing the instructions i_s and i_t. We have mostly shown what invariants are invalidated in §2 and §3. Here we explain one important case that is not clearly presented so far. If i_s is x `:=` e, all propositions about x_s are removed; and furthermore x is unconditionally added to the maydiff set, which may be dropped later by `auto` in case i_t executes an equivalent instruction. We proceed similarly for i_t.

The function $\texttt{add}(i_s, i_t, Q)$ strengthen the given post-invariant Q by adding new propositions obtained from the fact that i_s and i_t are just executed. We have mostly shown what invariants are newly added in §2 and §3. Here we explain one important case that is not clearly presented so far. If both i_s and i_t execute $p :=$ `alloc(..)` with equivalent arguments, then the register p is removed from the maydiff set.

Finally, the function $\texttt{auto}(Q)$ automatically applies predefined sequences of inference rules to the given post-invariant Q to get a stronger post-invariant. First, `auto` in our validator applies inference rules regarding phinodes and loops, as discussed in §3.4. Second, `auto` applies the `reduce_maydiff` rule to the register just defined in order to

	Compiler (Covered)	Proof Generation
`instcombine`	799	4950
`gvn`	1496	1207
`sroa`	989	1872
Library	-	17020 (auto-gen'd: 11007)

Figure 4. Lines of Compiler and Proof Generation Code

remove it from the maydiff set in case the same instruction is executed in source and target.

For example, suppose that i_s and i_t are `x := y + 1` and `y` is not in the maydiff set of the pre-invariant. Even so, the function `add` unconditionally adds `x` to the maydiff set of the post-invariant. However, the function `auto` removes the register `x` from the maydiff set by applying `reduce_maydiff(x)`.

Note that the functions `check`, `remove` and `add` are parts of TCB. For the function `auto`, only the inference rules used by `auto` are parts of TCB, but the automation algorithm of `auto` itself is not.

5.3 Inference Rules

The function `apply` : *InfRule* $\times$ *Inv* $\rightarrow$ *Inv* defines how the inference rules work. The set *InfRule* consists of inference rule names with arguments and the function `apply` is defined separately for each inference rule. In order to extend our validator with new inference rules, we just need to add more rules in the set *InfRule* and add to `apply` the definitions (*i.e.,* functions from *Inv* to *Inv*) of the newly added rules.

The `apply` function is a part of TCB and should be sound in the sense that for any inference rule r, any source and target program states satisfying an invariant I should also satisfy the invariant `apply`(r, I).

6. Results

6.1 Development

Our validator targets on 146 micro-optimizations in the `instcombine` pass, the main GVN-PRE algorithm in the `gvn` pass, and the register promotion algorithm in the `sroa` pass in LLVM 3.7.1. We just added the proof generation code to the original compiler without changing any existing code.

Figure 4 shows the number of lines of the compiler code in C++ for which we inserted the proof generation code. We separately counted the number of lines of our library for writing proof generation code. 64.7% of the library is about printing commands for inference rules and are automatically generated from 1,835 lines of code in a custom DSL.

The proof generation code is long compared to the covered compiler code, and in particular, it is as twice as long for register promotion in `sroa`. This is mainly because we needed to reimplement the compiler's logic in the proof generation code in order to keep the original compiler code untouched. Another reason is that the proof generation code contains many boilerplate code. We expect it can be reduced significantly.

The development of our validator took approximately 30 person-months in total. It took 5 months to upgrade and stabilize Vellvm, such as upgrading memory model and fixing a memory leak. It took 1 month to implement the validator, and 3 months to formally verify it. It took 2 months to implement the C++ library for proof generation. It took 2 month to prepare test infrastructure for the validator. It took 4 months to validate `instcombine`, 4 months to validate `gvn`, and 9 months to validate `sroa`. It took longer to validate optimization passes than expected, mainly because the compiler code is not designed in consideration of verification.

6.2 Performance Evaluation

We evaluate the performance of our validator in our experiments, in which we validated the compilation of the SPEC CINT2006 C Benchmarks [5] and other 5 open-source C projects (the biggest benchmarks used in [17][7]), totalling 3.9 million lines of C code. We omitted 4 files from the benchmarks: 3 of them contain instructions currently not supported in Vellvm, including the `indirectbr` instruction, and one is simply too big (151MB) for our validator to handle. We conducted the experiments in a Linux workstation with Intel Xeon E5-2630 CPU (2.6GHz, 12 cores with hyper-threading), 128GB RAM, and 1TB SSD (Samsung 850 PRO).

Figure 5 summarizes the experimental results for each optimization pass. In the experiment, we compiled each benchmark program using the original and the modified compilers with the `-O2` option. We counted the number of the generated validation proofs (**#V**), those proofs our validator failed to validate (**#F**), and those currently not supported (**#NS**). Everything else (**#V** $-$ **#F** $-$ **#NS**) is successfully validated. We also measured the time spent to perform each optimization in the original compiler (**Comp**), to perform each optimization and calculate validation proofs in the modified compiler (**PCal**), to print the source and the target programs and the proofs to files (**Print**), and to validate the generated proofs by the validator (**Vali**). When measuring the times, we maximized the CPU utilization by executing 24 jobs in parallel (because the compiler and the validator are single-threaded programs), measured the CPU time (User+System) used by each job, and summed up them to get the total.

Out of 1,191,063 validations in total, 993,157 (83.4%) are successfully validated. All 254 (0.02%) failures are due to compiler bugs: 247 are due to two bugs in `gvn`, and 7 are due to a bug in `sroa`. We present the three compiler bugs in the supplementary material [1, §A].[8] The other 197,652 (16.6%) translations are currently not supported in our validator, among which 191,819 (97.0%) use instructions not supported in Vellvm: vector operations (181,640, 91.9%), debug attributes (8,485, 4.3%) and atomic fences (1,694, 0.9%); 2838 (1.4%) use the alias analysis module of LLVM;

[7] We omitted Linux, since it is currently not compiled with LLVM. See [3] for more details.

[8] We do not identify the bugs for the double-blind review.

	LOC	Result									Time (sec.)											
		instcombine			gvn			sroa			instcombine				gvn				sroa			
		#V	#F	#NS	#V	#F	#NS	#V	#F	#NS	Comp	PCal	Print	Vali	Comp	PCal	Print	Vali	Comp	PCal	Print	Vali
400.perlbench	168.16K	59753	0	7357	11882	17	0	1745	0	1	5.66	36.44	3877.23	370.05K	1.37	2.25	135.36	138.34K	0.25	55.30	123.41	94.32K
401.bzip2	8.29K	4545	0	1865	1681	0	0	90	0	0	0.39	4.20	135.84	2.82K	0.09	0.23	7.39	2.58K	0.02	3.88	6.33	35.47K
403.gcc	517.52K	140730	0	5076	36999	20	13	5263	0	5	18.71	155.13	15376.14	662.28K	5.49	7.94	378.13	121.25K	0.71	157.35	317.45	433.81K
429.mcf	2.69K	492	0	57	149	0	0	24	0	0	0.05	0.13	2.51	0.03K	0.01	0.01	0.23	0.01K	< 0.01	0.29	0.46	0.02K
433.milc	15.04K	3840	0	517	2046	0	0	235	0	2	0.48	1.04	40.49	0.32K	0.11	0.14	4.54	0.20K	0.03	2.36	5.41	0.34K
445.gobmk	196.24K	16324	0	1724	7176	0	56	2641	0	1	3.05	8.41	1332.85	12.82K	0.70	0.97	42.46	5.58K	0.19	37.04	134.19	51.59K
456.hmmer	35.99K	11123	0	3433	3538	3	3	558	0	0	1.49	4.02	178.75	1.16K	0.32	0.39	29.32	0.63K	0.07	10.66	23.44	2.88K
458.sjeng	13.85K	3885	0	297	1751	0	1	130	0	0	0.60	1.84	73.41	1.17K	0.15	0.20	5.79	0.44K	0.02	3.19	5.52	0.97K
462.libquantum	4.36K	1052	0	793	357	0	265	123	0	79	0.15	0.31	9.18	0.05K	0.02	0.03	1.44	0.02K	0.01	0.68	1.56	0.01K
464.h264ref	51.58K	22885	0	5529	12858	27	4	532	1	0	2.76	21.41	1243.88	16.60K	0.74	1.54	122.60	13.38K	0.10	27.09	60.17	84.35K
470.lbm	1.16K	174	0	51	77	0	0	19	0	0	0.03	0.07	1.59	0.01K	0.01	0.01	0.08	< 0.01K	< 0.01	0.38	0.60	0.54K
482.sphinx3	25.09K	6280	0	1056	1652	0	3	364	0	0	0.77	1.91	61.69	0.58K	0.14	0.18	3.32	0.22K	0.04	4.01	8.11	0.69K
sendmail-8.15.2	138.68K	14172	0	411	4647	0	120	536	0	403	2.15	7.22	773.93	11.48K	0.52	0.64	38.63	5.38K	0.13	7.96	19.89	0.07K
emacs-25.1	463.54K	113316	0	9606	28730	15	35	5149	0	4	15.37	62.54	11343.41	130.23K	2.93	4.62	479.52	48.04K	0.65	150.74	346.35	222.95K
python-3.4.1	486.38K	89623	0	13419	28014	117	51	8784	0	89	15.92	54.03	6793.55	49.99K	3.74	5.28	390.16	27.47K	0.87	94.07	281.39	39.69K
gimp-2.8.18	1004.20K	151905	0	44676	38282	22	438	19448	6	528	29.15	79.67	3543.97	20.13K	5.03	7.27	215.32	8.46K	1.55	124.29	466.41	26.41K
ghostscript-9.14.0	797.65K	247485	0	83541	68166	26	6969	12996	0	9176	27.72	95.39	8167.96	68.77K	7.64	11.22	659.51	40.12K	2.11	59.27	188.68	1.50K
LLVM nightly tests	1358.76K	715212	0	353960	118375	49	310	17980	3	291	57.36	361.59	33350.68	685.17K	11.82	20.71	1227.30	144.46K	2.04	375.85	831.41	660.52K
Total	5289.18K	1602796	0	533368	366380	297	8268	76617	10	10579	181.81	895.36	86307.04	2033.66 K	40.81	63.65	3741.10	556.56K	8.81	1114.43	2820.80	1656.10K

Figure 5. Experimental Results

2200 (1.1%) alter type declarations; 693 (0.4%) require deeper analysis on functions such as read-only function analysis; and 102 (< 0.1%) are beyond the scope of the main GVN-PRE algorithm.

Proof generation and validation are slow compared to compilation. In particular, on average, proof calculation took 4.3 times for `instcombine`, 1.5 times for `gvn`, and 108.2 times for `sroa`, compared to the original compilation time.

Since our proof generation code does not have bigger time complexity than the optimization algorithm itself has, we believe we can significantly reduce the time for generating and validating proofs. There is plenty of room for improvement in calculating proofs. For example, the proof calculation code creates unnecessarily many C++ objects and auxiliary data. In particular, `sroa` calculates too verbose proofs (the average size of a proof is 343.42KB), mainly because we validated the whole step of a register promotion translation at once. There are opportunities of splitting the whole proof into many smaller ones avoiding the problem of broken SSA (see §4.3), which we believe will reduce the proof generation time significantly. Furthermore, we can pass programs and proofs from the compiler to the validator faster, (i) by compressing the data by printing only difference between the source and the target programs and making the proofs more terse, and (ii) by writing data to RAM instead of SSD. Also, the validator is naively implemented in order to make verification of the validator easier. Finally, we can massively parallelize the validations in the clouds, as each validation job is independent from each other.

6.3 Verification

In order to reduce our validator's TCB, we formally verified major parts of our validator in the Coq proof assistant.

Semantics We use the formal semantics of LLVM IR from the Vellvm project [26]. Vellvm used the CompCert memory model [14] version 1.9 and we upgraded it to version 2.4 in order to use the notion of *permission* in the LLVM semantics. We also added the `switch` instruction to the formalization of LLVM IR, which is used by 50.8% of our benchmark C projects used for performance evaluation.

Vellvm still lacks the support for various features in LLVM, including vector operations, atomic operations for concurrency, and attributes such as `noalias` and `readonly`. Furthermore, casts between integers and pointers are not properly formalized in Vellvm and the CompCert memory model. However, the idea of Kang *et al.* [11] to model integer-pointer casts is applicable to our memory model, which we leave as future work.

Finally, the *undef* value, a weaker notion than poison, of LLVM IR is simply formalized as poison in Vellvm. In fact, it is well-known in the LLVM community that the semantics of `poison` together with `undef` is seriously broken [4]. Due to the lack of a weaker notion of *undef* semantics, some optimizations, such as loop unswitching, are not valid in our model as well as in CompCert. Further research is needed to solve this problem.

Verification of Validator We proved in Coq the correctness of the post-condition generator of our validator using simulation relations similar to those used in CompCert, currently leaving many admits on minor details. The main theorem states that if our validator validates a proof of a translation, the source and target programs are related by a *simulation* relation. We also proved the adequacy of simulation relations (*i.e.,* simulation implies behavioral refinement). Our correctness result is the same as that of CompCert.

Inference Rules We installed 194 inference rules for our experiment. None of them are verified but we believe they are clearly sound because they are all simple rules. In fact, we proved about 30 inference rules sound in Coq for an early version of our development, which we have not ported to the current version yet.

7. Discussion

7.1 Related Work

There has been many proposals for compilation verification/validation, but our work is the first verified approach to mainstream compilers.

(Foundational) proof carrying code (PCC) [16, 6] is a (verified) proof system for validating safety properties of a program. We are inspired by PCC's use of proofs for validating program properties, but we validated the relational property of semantic preservation between source and target programs, rather than unary properties of a single program.

Tristan *et al.* [20] and Stepp *et al.* [19] developed translation validators for LLVM optimization passes, including dead code elimination, GVN-PRE, constant propagation, and loop invariant code motion. We are also inspired a lot by these approaches. However, these are differ from our approach in several aspects. First, their validators are inherently incomplete, since they are not given any extra information from compilers. Hence they failed to validate about 20% of programs in the benchmark. Furthermore, they do not cover register promotion, which quires complex memory reasoning, nor formally verify their validators.

CompCert [13] is the first formally verified optimizing C compiler. However, CompCert implements a relatively small number of optimizations compared to mainstream compilers such as LLVM. Also, CompCert is implemented in Coq [2] and run in OCaml, which is a purely functional language, and thus its compilation time is relatively slow compared to the mainstream compilers written in C++. We think the verified compiler approach is not very adequate for mainstream compilers because the verification cost is high and the compilation time is slow.

Zhao *et al.* [26, 25] implemented and verified the `vmem2reg` pass in Coq for the Vellvm project. The `vmem2reg` pass performs register promotion, but its algorithm is significantly simplified compared to that in LLVM.

Translation validators are verified for various optimizations in CompCert. Tristan and Leroy verified ones for instruction scheduling [21], lazy code motion [23], software pipelining [22]; Rideau and Leroy, register allocation [18]; Barthe *et al.* [7], SSA transformation; and Demange *et al.* [10], GVN and sparse conditional constant propagation (SCCP). These validators provide the same level of correctness guarantee as verified compilation. Compared to our approach, the difference is that these validators are highly specialized for the target optimizations, without sharing any common validator like our ERHL validator.

Lopes *et al.* [15] presented Alive, a domain-specific language for writing peephole optimizations. Using Alive, one can either prove the correctness of an optimization algorithm using SMT solvers, or find a counterexample. The authors ported 300 micro-optimizations in `instcombine` to Alive, and in doing so they found 8 bugs in `instcombine`. While practical, Alive is limited to supporting only peephole optimizations that are performed inside a single block (*i.e.,* programs with linear control flow). Furthermore, Alive relies on SMT solvers and makes simplifying assumptions on the LLVM semantics, and thus provides a lower level of correctness guarantee than that of compilation verification performed in Coq.

CSmith [24, 8] and EMI [12] are random testing systems for validating C compilers. They improved the reliability of mainstream C compilers by finding hundreds of bugs in them. However, testing does not assure the absence of bugs.

7.2 Future Work

There are a number of interesting issues remaining for future work.

Changing Control-Flow Graph So far we assumed that the source and the target programs can be aligned by inserting logical no-op instructions. While this assumption applies to the majority of LLVM optimization passes (see the supplementary material [1, §B] for more details), some important optimizations, including loop unrolling and loop unswitching, are beyond the reach of the current paper. We believe we can lift the assumption by generally mapping the source and the target program points for the validator.

Supporting Analysis Passes Currently our validator does not support analysis passes, including alias analysis, read-only function analysis, and memory dependence analysis. We believe we can support them by adding appropriate predicates and inference rules in our validator.

Supporting Missing Features As discussed in §6.3, it is an interesting future work to support missing features in Vellvm.

Validating Frontend & Backend We focused on the IR-to-IR optimization passes, leaving the frontend (C to IR) and backend (IR to assembly) passes as future work.

8. Conclusion

In this paper, we propose a practical approach to compilation verification that applies to mainstream compilers without sacrificing the compilation performance.

The post validation does not need to be performed every time we compile because it consumes huge computing resources. We think it would most useful to perform post validation only before deploying the software or sometimes in debugging to make sure the absence of compiler bugs.

Also we believe that it would save a lot of efforts if we develop a compiler together with the validator because we can design a compiler in a way that validation become easier.

References

[1] Anonymized development and supplementary material.

[2] The coq proof assistant. https://coq.inria.fr/. Accessed: 2016-11-15.

[3] LLVM Linux. http://llvm.linuxfoundation.org. Accessed: 2016-11-15.

[4] RFC: Killing undef and spreading poison. http://lists.llvm.org/pipermail/llvm-dev/2016-October/106182.html. Accessed: 2016-11-15.

[5] The spec cint2006 benchmark. https://www.spec.org/cpu2006/CINT2006/. Accessed: 2016-11-15.

[6] Andrew W. Appel. Foundational proof-carrying code. LICS '01.

[7] Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM Trans. Program. Lang. Syst.*, 36(1).

[8] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. PLDI '13.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4).

[10] Delphine Demange, David Pichardie, and Léo Stefanesco. Verifying fast and sparse SSA-based optimizations in Coq. CC '16.

[11] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A formal C memory model supporting integer-pointer casts. PLDI '15.

[12] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. PLDI '14.

[13] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. POPL '06.

[14] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, 2012.

[15] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. PLDI '15.

[16] George C. Necula. Proof-carrying code. POPL '97.

[17] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. Global sparse analysis framework. *ACM Trans. Program. Lang. Syst.*, 36(3).

[18] Silvain Rideau and Xavier Leroy. Validating Register Allocation and Spilling. CC '10.

[19] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for llvm. CAV'11.

[20] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for llvm. PLDI '11.

[21] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. POPL '08.

[22] Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. POPL '10.

[23] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. PLDI '09.

[24] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. PLDI '11.

[25] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. PLDI '13.

[26] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. POPL '12.

A. Compiler Bugs

During the development of our validator, we discovered three compiler bugs in LLVM. In this section we report what are they, and how we found them.

Two Bugs in GVN-PRE regarding the Poison Value In our experiment, our validator failed to validate hundreds of `gvn` translations, which turn out to be due to two compiler bugs regarding the poison value and the `getelementptr` (GEP) instruction. GEP is an instruction that basically performs pointer arithmetic, with an optional `inbounds` flag that makes the result `poison`ed if the result goes out of array bounds. As an example, consider the following program:

```
int a[2];
p = a + 4 inbounds;  // GEP
q = a + 4;           // GEP
```

Here, GEP instructions are represented simply as pointer arithmetic. Since `a + 4` is out of bounds, `p` gets `poison`, while `q` is a normal pointer value pointing to outside of the array `a`.

Even though the values of `p` and `q` differ, the main GVN algorithm in the `gvn` pass classified them as equivalent, incorrectly replacing `q` with `p`. We reported this bug to the LLVM bug tracker, and it is fixed in LLVM 3.9.0.

We found a similar bug in the PRE algorithm of the `gvn` pass and reported it also to the bug tracker, and it is confirmed.

A Bug in Register Promotion regarding Loops We analyzed the implementation of register promotion in LLVM before validating it, and surprisingly, we found a 7-year-old bug regarding loops. For *e.g.,* LLVM performed the following translation:

```
p := alloc();          p := alloc();
loop {                 loop {
    r := *p;                r := undef;
    *p := 42;      ⇝        (lnop;)
    ...                    ...
}                      },
```

provided that all the accesses to `p` in the source are within the loop. LLVM essentially failed to recognize the loop structure of the source program, and replaced the load from `p` with `undef` (a variant of `poison`), which is incorrect since `r` becomes 42 from the second iteration of the loop in the source, while it is still `undef` in the target. Note that a phinode should be inserted at the beginning of the block in a correct translation. We reported this bug to the `llvm-dev` mailing list, and the bug was subsequently fixed in LLVM 3.8.0.

B. Classification of LLVM `-O2` Passes

LLVM's `-O2` compiler flag consists of 69 analysis and transformation passes.

We believe the following 46 (66.7%) passes can be supported by extending the current validator: `aa`, `adce`, `alignment-from-assumptions`, `assumption-cache-tracker`, `barrier`, `basicaa`, `bdce`, `block-freq`, `branch-prob`, `correlated-propagation`, `demanded-bits`, `domtree`, `dse`, `early-cse`, `elim-avail-extern`, `float2int`, `forceattrs`, `functionattrs`, `globaldce`, `globalopt`, `globals-aa`, `gvn`, `inferattrs`, `instcombine`, `instsimplify`, `lazy-block-freq`, `lazy-value-info`, `lcssa`, `licm`, `loop-load-elim`, `mem2reg`, `memcpyopt`, `memdep`, `mldst-motion`, `opt-remark-emitter`, `profile-summary-info`, `reassociate`, `rpo-functionattrs`, `sccp`, `speculative-execution`, `sroa`, `strip-dead-prototypes`, `targetlibinfo`, `tbaa-scoped-noalias`, `tti`, `verify`.

We believe we can further support the following 19 (27.5%) passes that changes control-flow graphs, by generally mapping the source and the target program points: `basiccg`, `constmerge`, `indvars`, `jump-threading`, `loop-accesses`, `loop-deletion`, `loop-distribute`, `loop-idiom`, `loop-rotate`, `loop-simplify`, `loop-unroll`, `loop-unswitch`, `loop-vectorize`, `loops`, `pgo-icall-prom`, `prune-eh`, `scalar-evolution`, `simplifycfg`, `slp-vectorizer`.

The following 4 (5.8%) passes require further generalization, which we leave as future work: `deadargelim`, `inline`, `ipsccp`, `tailcallelim`.

C. Micro-Optimizations in `instcombine`

We validated 146 micro-optimizations in `instcombine`.

It is required to insert `lnop` instructions (§3.1) for validating the following 30 micro-optimizations: `dead-code-elim`, `fold-phi-bin`, `fold-phi-bin-const`, `load-load`, `add-mask`, `add-dist-sub`, `sdiv-mone`, `udiv-zext`, `urem-zext`, `or-or2`, `select-bop-fold`, `or-xor4`, `trunc-onebit`, `zext-trunc-and-xor`, `zext-xor`, `icmp-eq-xor-not`, `icmp-ugt-and-not`, `load-store`, `dead-store-elim`, `select-icmp-eq-xor1`, `select-icmp-eq-xor2`, `select-icmp-ne-xor1`, `select-icmp-ne-xor2`, `select-icmp-sgt-xor1`, `select-icmp-sgt-xor2`, `select-icmp-slt-xor1`, `select-icmp-slt-xor2`, `sext-trunc-ashr`, `and-or-const2`, `and-xor-const`.

The following 3 micro-optimizations rely on simple alias analysis (§3.2): micro-optimizations: `load-load`, `load-store`, `dead-store-elim`.

It is required to encode relational predicates with ghost registers (§3.3) for validating the following 4 micro-optimizations: `load-load`, `load-store` `fold-phi-bin`, `fold-phi-bin-const`.

Note that it is additionally required to handle phinodes and loops as discussed in §3.4, for validating `fold-phi-bin` and `fold-phi-bin-const`.

The following 5 micro-optimizations essentially replaces a poison value with a concrete value (§3.5): `and-undef`, `or-undef`, `xor-undef`, `shift-undef1`, `shift-undef2`

The following 111 (76.0%) is validated without using the ideas presented in §3: `bop-associativity`, `sub-add`, `add-sub`, `add-comm-sub`, `mul-bool`, `mul-neg`, `add-shift`, `add-signbit`, `sub-remove`, `add-onebit`, `add-zext-bool`, `add-const-not`, `add-select-zero`, `sub-mone`, `sub-onebit`, `sub-shl`, `sub-sdiv`, `sub-const-not`, `sub-const-add`, `mul-mone`, `mul-shl`, `div-sub-rem`, `srem-neg`, `and-de-morgan`, `or-or`, `or-xor`, `or-xor2`, `add-xor-and`, `add-or-and`, `sub-sub`, `sub-or-xor`, `and-or`, `and-same`, `and-zero`, `and-not`, `and-mone`, `or-not`, `or-mone`, `or-zero`, `or-same`, `or-and`, `or-xor3`, `or-and-xor`, `bitcast-bitcast`, `ptrtoint-bitcast`, `fpext-fpext`, `fptosi-fpext`, `fptoui-fpext`, `bitcast-inttoptr`, `sext-sext`, `sitofp-sext`, `trunc-trunc`, `sext-zext`, `uitofp-zext`, `zext-zext`, `xor-same`, `xor-zero`, `bitcast-sametype`, `bitcast-fptosi`, `bitcast-fptoui`, `bitcast-ptrtoint`, `bitcast-trunc`, `bitcast-sext`, `bitcast-zext`, `bitcast-fpext`, `bitcast-fptrunc`, `bitcast-sitofp`, `bitcast-uitofp`, `inttoptr-bitcast`, `sext-bitcast`, `sitofp-bitcast`, `trunc-bitcast`, `uitofp-bitcast`, `zext-bitcast`, `fpext-bitcast`, `fptosi-bitcast`, `fptoui-bitcast`, `fptrunc-bitcast`, `trunc-zext`, `trunc-sext`, `fptrunc-fpext`, `inttoptr-ptrtoint`, `sitofp-zext`, `zext-trunc-and`, `icmp-swap`, `icmp-ne-xor`, `icmp-ult-and-not`, `icmp-sgt-and-not`, `icmp-slt-and-not`, `icmp-uge-or-not`, `icmp-ule-or-not`, `icmp-sge-or-not`, `icmp-sle-or-not`, `icmp-eq-sub`, `icmp-ne-sub`, `icmp-eq-srem`, `icmp-ne-srem`, `icmp-eq-add-add`, `icmp-eq-sub-sub`, `icmp-eq-xor-xor`, `icmp-ne-add-add`, `icmp-ne-sub-sub`, `icmp-ne-xor-xor`, `select-icmp-eq`, `select-icmp-ne`, `shift-zero1`, `shift-zero2`, `select-icmp-gt-const`, `select-icmp-lt-const`, `ptrtoint-inttoptr`, `and-or-not1`.