CRELLVM: Verified Credible Compilation for LLVM

Jeehoon Kang∗ Yoonseung Kim∗ Youngju Song∗
{jeehoon.kang, yoonseung.kim, youngju.song}@sf.snu.ac.kr
Seoul National University, Korea

Sungkeun Cho Joonwon Choi
skcho@ropas.snu.ac.kr joonwonic@mit.edu
Seoul National University, Korea MIT, USA

Juneyoung Lee Sanghoon Park
Mark Dongyean Shin Yonghyun Kim
{juneyoung.lee, sanghoon.park}@sf.snu.ac.kr
{dongyean.shin, yonghyun.kim}@sf.snu.ac.kr
Seoul National University, Korea

Chung-Kil Hur† Kwangkeun Yi
gil.hur@sf.snu.ac.kr kwang@ropas.snu.ac.kr
Seoul National University, Korea

Abstract
Production compilers such as GCC and LLVM are large complex software systems, for which achieving a high level of reliability is hard. Although testing is an effective method for finding bugs, it alone cannot guarantee a high level of reliability. To provide a higher level of reliability, many approaches that examine compilers’ internal logics have been proposed. However, none of them have been successfully applied to major optimizations of production compilers.

This paper presents Crellvm: a verified credible compilation (or equivalently, verified translation validation) framework for LLVM, which can be used as a systematic way of providing a high level of reliability for major optimizations in LLVM. Specifically, we augment an LLVM optimizer to generate translation results together with their correctness proofs, which can then be checked by a proof checker formally verified in Coq. As case studies, we applied our approach to two major optimizations of LLVM: register promotion (mem2reg) and global value numbering (gvn), having found four miscompilation bugs (two in each).

1 Introduction
Production compilers such as GCC and LLVM are large complex software systems, for which achieving a high level of reliability is hard. Their complexity comes in two fold. First, to generate efficient target code, they perform various complex optimizations. Second, to consume less time and memory during compilation, they are usually written in C/C++ using sophisticated data structures. Due to such complexity, it is hard to make mainstream compilers very reliable.

Although testing is an effective method for finding bugs, it alone cannot guarantee a high level of reliability. For example, recent random testing tools such as CSmith [41] and EMI [16] have shown their effectiveness by finding hundreds of bugs in GCC and LLVM. However, they are hardly sufficient to provide a high level of reliability guarantee since they treat compilers as black boxes without examining their internal logics.

In order to provide a higher level of reliability, many approaches that examine compilers’ internal logics have been proposed, none of which, however, have successfully applied to major optimizations of production compilers. For example, verified compilers such as CompCert [17] are written and verified inside a proof assistant such as Coq and thus guarantee their formal correctness. However, this approach is not readily applicable to production compilers since they are mostly written in C/C++, not in the language of a proof assistant. As another example, Alive [20] is a domain-specific language (DSL) in which one can manually write a compiler’s optimization logic and automatically verify its correctness or else generate a counter example. Though this approach has been successfully applied to LLVM, its application is limited to peephole optimizations because it is hard to faithfully translate the implementation of complex optimizations into Alive and, more importantly, Alive does not support cyclic control flows such as loop. As the last example, the credible compilation [8, 23, 24, 32] and verified translation validation [7, 11, 31, 38–40] approaches augment compilers to generate translation results together with their correctness proofs, which can then be checked by a (verified) proof checker. Since a correctness proof is generated and checked at each compilation time, it provides a formal correctness guarantee for the particular translation or else find a bug (either in the compiler code or in the proof generation code). However, there has been only a preliminary attempt to apply these approaches to production compilers so far. (See §8 for detailed comparison.)

This paper presents Crellvm: a verified credible compilation framework for LLVM, which can be used as a systematic way of providing a high level of reliability for major optimizations in LLVM. Specifically:

1. We design and develop a logic and its proof checker for reasoning about LLVM optimizations, called Extensible Relational Hoare Logic (ERHL), in the proof assistant Coq.
2. We fully verify a semantics preservation result for our proof checker in the style of CompCert using the Coq formalization of LLVM IR (Intermediate Representation) from the Vellvm project [43].

3. As case studies, we wrote proof generation codes (213 and 413 SLOC\(^1\) in C++) for two major optimizations: register promotion (mem\(_2\)reg pass) and global value numbering with partial redundancy elimination (gvn pass). Then we performed validation of the two optimizations for standard benchmarks and five large open-source projects.

4. As a result, we found four miscompilation bugs (two in each optimization). This result is notable because previous systematic approaches including random testing have found any miscompilation bugs in the mem\(_2\)reg and gvn passes. Moreover, except for the two bugs we have reported, we found only one confirmed miscompilation bug for mem\(_2\)reg in the LLVM bug tracker history\(^2\), which was reported in 2005 — in the early days of LLVM. Also, one of the bugs we found in mem\(_2\)reg had not been found for 7 years even though it could miscompile realistic programs (see Appendix B\(^3\)).

1.1 Overview of Crellvm

Framework  The framework of Crellvm works as follows. First, as shown in Fig. 1, we separate the compilation and validation phases. For compilation, as depicted in the left side of Fig. 1, we use the original optimizer to translate the source IR code src\(_{.11}\) to the target IR code tgt\(_{.11}\). After the compilation, we can conduct validation, as depicted in the right side of Fig. 1. For this, we first run the optimizer extended with a proof generation code that produces the target tgt\(_{'.11}\) together with the proof Proof. Then the proof checker validates Proof to see whether src\(_{.11}\) is correctly translated to tgt\(_{'.11}\). If the validation fails, we can see a logical reason for the failure, with which we can find a bug either in the compiler or in the proof generation code. If the validation succeeds, we finally compare tgt\(_{.11}\) and tgt\(_{'.11}\) using the LLVM IR comparison tool llvm-diff.

There are two points to note about the framework. First, we explicitly distinguish the original compiler from the proof generating one because the former is what the users will use and the latter is what the compiler developers will use to increase the reliability of the former. Second, llvm-diff essentially performs alpha-equivalence checking, which is necessary because while tgt\(_{.11}\) may have unnamed IR registers, tgt\(_{'.11}\) has explicit names for all registers for a proof generation purpose.

**ERHL and Proof Checker**  For validation in Crellvm, we develop ERHL, which is a variant of relational Hoare logic [8] specialized for LLVM IR. The logic and its proof checker is extensible because (i) the logic can be extended with any custom inference rules and (ii) the proof checker can be extended with any custom automation functions that try to fill in the gaps in incomplete proofs by automatically finding appropriate inference rules, like the auto tactic in Coq.

**Verification of Proof Checker**  In the Crellvm framework, the TCB (Trusted Computing Base) includes only the proof checker, the equality checker (llvm-diff) and custom inference rules. In particular, the proof generation code in the compiler is not a part of TCB because any incorrect proof would be invalidated by the proof checker.

We further remove the proof checker and inference rules from the TCB by implementing and verifying them in Coq. Though we currently use the (unverified) standard llvm-diff tool for comparing IR programs, it would be also possible to implement and verify it in Coq.

Note that verification of the proof checker and inference rules matters in practice. First, we found various corner case bugs in our proof checker during its verification. Second, we also found one of our two mem\(_2\)reg bugs during the verification of inference rules. See the example below.

\[
p := \text{alloca}(); r := *p \\
\text{foo}(r) \rightarrow \text{foo}(1 / ((\text{int})G - (\text{int})G)) \\
i *p := 1 / ((\text{int})G - (\text{int})G)
\]

Here G is the constant address of a global variable.

To see why this translation is incorrect, suppose that the function foo\((r)\) ignores \(r\) and repeatedly prints out 0 without returning to the caller. Then division-by-zero never happens in the source program, while it does in the target. The problem here is that the mem\(_2\)reg pass assumes that constant expressions never raise any undefined behavior such as division-by-zero, which is not true since \(1 / ((\text{int})G - (\text{int})G)\) forms a valid constant expression in LLVM. Following the logic of mem\(_2\)reg, we also added such a custom inference rule, which we found unsound during the verification of the rule.

It is important to note that all the programs in this paper represent LLVM IR programs and we just use C syntax to help understanding. For example, the source program in the above transformation is undefined as a C program but well-defined

---

\(^1\)SLOC stands for significant lines of code i.e., ignoring spaces and comments.

\(^2\)We examined all miscompilation reports in https://bugs.llvm.org.

\(^3\)This technical appendix is submitted as supplementary material.
as an IR program. Thus, the transformation is only unsound as an IR-to-IR transformation. The LLVM community considers such an IR-to-IR miscompilation as a definite bug even when it does not cause any C-to-Assembly miscompilation since it can potentially cause an end-to-end miscompilation for other source languages such as Swift and Rust.

**Results** We wrote proof generation codes for Register Promotion in the mem2reg pass and for GVN-PRE in the gvn pass; and also partly for Loop Invariant Code Motion in the licm pass, and 139 micro-optimizations in the instcombine pass in order to demonstrate the generality of ERHL. We then conducted validation of the optimizations for the SPEC CINT2006 C Benchmarks [4], LLVM nightly test suite, and five open-source projects: sendmail, emacs, python, gimp, and ghostscript, in total 5.3 million LOC in C. The validation failed at only those miscompilation caused by the four bugs we found.

We present the details of mem2reg validation in §3 and gvn validation in Appendix C.

### 1.2 Advantages of Crellvm over Testing

Crellvm checks whether optimizations are performed by correct reasoning, while testing simply checks results of the test programs. This can make difference as follows.

First, an optimization performed by incorrect reasoning may be still correct for most programs including all the test programs. In this case, testing cannot uncover the bug, while Crellvm can because it checks the underlying reasoning. For example, we found our first mem2reg bug in this situation, which had not been found for 7 years.

Specifically, the following optimization shows the bug.

```plaintext
p := alloca()
loop {
  r := *p; foo(r); *p := 42$ ~$ foo(undef)
}
```

In the source program, at the first iteration, the register `r` is assigned the undef value\(^4\) because the memory cell `*p` is uninitialized, while at the second iteration, `r` is assigned 42. On the other hand, in the target program, `r` is always replaced by undef. Of course, this is performed due to faulty reasoning of mem2reg\(^5\). However, it can be correct for most programs because `r` can be sometimes undef and thus `foo(r)` is likely to ignore `r` (e.g., by taking an operation like `r & 0x0`). Indeed this is what happened in the SPEC benchmark and why the bug had been hidden for such a long time. Note that it does not mean that this bug does not matter in practice because it could well cause trouble in the real world (see Appendix B).

Second, a potential flaw introduced by miscompilation may not be exploited by the current compiler and silently disappear during the compilation. Also in this case, Crellvm can detect the bug because it checks the underlying reasoning. For example, we found the two gvn bugs in this situation, which had not been found for 8 years. Note that the two bugs are caused by the same reason but we counted them as two because they appear in two separate places.

Specifically, the following optimization shows the bug.

```plaintext
q1 := (p + 10) inbounds q1 := (p + 10)
q2 := (p + 10)
bar(q1, q2) ~ bar(q1, q1)
```

In the source program, `(p + 10)` inbounds\(^6\) is defined to be undef\(^7\) when the index 10 is out of the bounds of `p`, while `(p + 10)` is always defined to be the computed address. Thus replacing `q2` with `q1` introduces more undefinedness, which is incorrect because it can be potentially exploited by subsequent optimizations. However, so far the LLVM compiler has not exploited such undefinedness, thereby causing no observable misbehaviors. Indeed this miscompilation happened many times during validation of the standard benchmarks but testing has failed to detect it.

### 2 Overview

In this section, we give a more detailed overview of how Crellvm works using the assoc-add optimization of the instcombine pass as a motivating example.

---

\(^4\)undefined is a special value representing undefinedness.

\(^5\)The bug was found in a special algorithm of mem2reg that only handles memory locations that are accessed only inside a single block.

\(^6\)This denotes the GetElementPtr (GEP) operation.

\(^7\)Technically speaking, it is defined to be poison but the difference is not important here.
2.1 Translation Example

We first give an example translation of the assoc-add optimization, which is shown in the shaded part of Fig. 2. Here \( y := \text{add } x \times 2 \) is replaced by \( y := \text{add } a \times 3 \) at line 20. This translation can be beneficial because after it, the register \( x \) may no longer be used and thus \( x := \text{add } a \times 1 \) at line 10 may be eliminated later. This translation is also sound because (i) the assertion “\( x = \text{add } a \times 1 \)” holds throughout lines 10-20, since the registers \( a \) and \( x \) are not redefined between line 10 and 20 thanks to the Static Single Assignment (SSA) property \([10]\); and (ii) from this, we can infer that \( x \times 2 = \text{add } (\text{add } a \times 1) 2 = \text{add } a \times 3 \) holds at line 20.

2.2 Proof Validation

We now construct a proof for the assoc-add translation example and validate it in ERHL.

**ERHL Proof** A formal proof of the translation is given in the box of Fig. 2. Specifically, the proof consists of a set of assertions and a list of inference rules at each line. For example, at line 20, the set of assertions is \( \{ \text{MD}(\emptyset) \} \) and the list of inference rules is \( \text{assoc_add}(x_{src}, y_{src}, a_{src}, 1, 2) \).

This ERHL proof captures the assertion and the inference step of the intuitive reasoning above. First, the assertion \( \text{MD}(\emptyset) \) at every line states that every register contains the same value in the source and target program states. Second, the additional assertion \( x_{src} = \text{add } a_{src} \times 1 \) between line 10 and line 20 states that in the source state, the value of the register \( x \) is equal to the result of \( \text{add } a \times 1 \). Finally, the inference rules \( \text{assoc_add}(x_{src}, y_{src}, a_{src}, 1, 2) \) and \( \text{reduce\_maydiff}(y) \) at line 20 are those that need to be applied for validation at line 20. The details of the rules will be given later when we discuss the validation process.

**ERHL Assertions** Before we proceed to the validation of the proof, we discuss ERHL assertions in more details. An ERHL assertion is a triple \((S, T, M)\), where \( S \) is a set of assertions that should hold for the source state; \( T \) is for the target state; and \( M \) is an assertion relating the source and target states.

First, the source and target assertions, \( S \) and \( T \), can contain various forms of predicates. For example, \( x_{src} = \text{add } a_{src} 1 \) is a source assertion and \( x_{tgt} = \text{add } a_{tgt} 3 \) is a target assertion. Here and henceforth, \( x_{src} \) and \( x_{tgt} \) represent the values of the register \( x \) in the source and target states, respectively. Though we only use the equality predicate for assoc-add, we will introduce various other predicates later. It is important to note that we do not allow any general assertion relating the source and target states such as \( x_{src} = y_{tgt} + 1 \).

Second, the relational assertion \( M \) 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 target states, which we denote by \( \text{MD}(M) \):

\[
\text{MD}(M) \iff \forall x \notin M. \ x_{src} = x_{tgt}.
\]

Note that the maydiff set is the only form of relational assertion relating the source and target states.

Finally, every ERHL assertion implicitly imposes that the public parts of the source and target memories should be equivalent. More precisely, we use the CompCert-style memory injection relation \([19]\). Later we introduce predicates that allow private memory allocations that do not belong to the public part of the memory (see §3.2).

The main advantage of ERHL assertions is that we can use the standard algorithm of the (unary) Hoare logic to compute post assertions because ERHL assertions are mainly unary \( (i.e., \) only for the source state, or for the target state, not relating them) except for the maydiff set. Though mainly unary, ERHL assertions can indirectly encode general forms of relational assertions (see §3.2 for details).

**Proof Validation** The gray text in Fig. 2 shows the validation process performed by the ERHL proof checker, which proceeds as follows.

First, the proof checker checks that the initial assertion holds for all possible initial states. It accepts the initial assertion \( \{ \text{MD}(\emptyset) \} \) in Fig. 2 since the source and target states are initially equivalent.

Second, the proof checker checks whether the Hoare triple \( \{ P \} I_{src} \sim I_{tgt} \{ Q \} \) at each line is valid. This means that the assertion \( Q \) after the line holds for all program states resulted by executing the source and target instructions \( I_{src} \) and \( I_{tgt} \) at the line under any program states satisfying the assertion \( P \) before the line. In Fig. 2, we only explain validations at lines 10 and 20 in details because the others are trivial.

At line 10, the proof checker first computes a strong post-assertion, \( \{ x_{src} = \text{add } a_{src} 1, x_{tgt} = \text{add } a_{tgt} 1, \text{MD}(\emptyset) \} \). Using our post-assertion computation algorithm, here, the algorithm simply adds the equality predicates corresponding to the executed instructions. Then, the assertion after line 10, \( \{ x_{src} = \text{add } a_{src} 1, \text{MD}(\emptyset) \} \), follows from the computed strong post-assertion by a simple inclusion check.

At line 20, the checker also computes a strong post-assertion, \( \{ x_{src} = \text{add } a_{src} 1, y_{src} = \text{add } x_{src} 2, y_{tgt} = \text{add } a_{tgt} 3, \text{MD}(y) \} \). Here, the post-assertion computation adds the equality predicates corresponding to the executed instructions and also adds the register \( y \) to the maydiff set because the executed source and target instructions are not identical. Then, the proof checker applies the inference rules given by the proof. The rule \( \text{assoc\_add}(x_{src}, y_{src}, a_{src}, 1, 2) \) derives \( y_{src} = \text{add } a_{src} 3 \).
Algorithm 1 AssocAdd(F: Function)
A1: for l1: y := add (reg x) (const C2) in F do
A2: if FindDef(F, x) is l1: x := add (reg a) (const C1) then
A3: C := Simplify(add C1, C2)
A4: ReplaceAt(F, l2, y := add (reg a) (const C))
A5: Assn(xsrc = add asrc C1, l1, l2)
A6: Inf(assoc_add(xsrc, ysrc, asrc, C1, C2), l2)
A7: end if
A8: end for
A9: Auto(reduce_maydiff)

from xsrc = add asrc 1 and ysrc = add xsrc 2 by associativity:
(assoc_add(x, y, a, C1, C2))
\[
\begin{align*}
    x &= \text{add } a \ C_1 \\
    y &= \text{add } x \ C_2 \\
    C &= C_1 + C_2 \\
    \text{add } \{ y = \text{add } a \ C \}
\end{align*}
\]

The rule reduce_maydiff(y) removes the register y from the maydiff set because ysrc = add asrc 3, y tgt = add a tgt 3 and a is not in the maydiff set:

(\text{reduce_maydiff}(y, e))
\[
\begin{align*}
    ysrc &= e_{src} \\
    tgt &= e_{tgt} \\
    \text{no registers in } e \text{ are in the maydiff set}
\end{align*}
\]

\begin{align*}
\text{remove } y \text{ from the maydiff set}
\end{align*}

Then, the assertion after line 20, \{ MD(\emptyset) \}, easily follows by a simple inclusion check.

Finally, the proof checker checks whether the same observable events (i.e., the same sequence of system calls) are produced at each line. It is the case in Fig. 2 because at line 20, no observable events are produced; and 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. In particular, at line 21, the proof checker explicitly checks that the same value is passed to the function \text{foo} because the function may produce observable events.

2.3 Proof Generation

Now we explain how we generate proofs for assoc-add.

Algorithm Algorithm 1 shows the assoc-add optimization algorithm implemented in LLVM’s instcombine pass, which is presented in a rather functional style for presentation purposes. Specifically, AssocAdd(F) optimizes each function definition F as follows (ignore the boxes for now, which are the proof generation code).

[Line A1]: Find an instruction of the form l1: y := add x C2 with C2 constant. In Fig. 2, 20: y := add x 2 can be picked.

[Line A2]: Check if x is defined by an instruction of the form l1: x := add a C1 with C1 constant. Here, FindDef(F, x) finds the instruction that defines the register x.9 In Fig. 2, 10: x := add a 1 is picked. [Lines A3-A4]: If it is the case, compute the constant C = C1 + C2 and replace the instruction at l2 with y := add a C. In Fig. 2, the instruction at line 20 is replaced by y := add a 3.

Proof Generation Once we understand the assoc-add optimization algorithm, it is quite straightforward to write the proof generation code given in the boxes of Algorithm 1.

[Line A5]: Add the assertion xsrc = add asrc C1 at every line between l1 and l2. In Fig. 2, the assertion xsrc = add asrc 1 is added at every line between 10 and 20. [Line A6]: Add the inference rule assoc_add(xsrc, ysrc, asrc, C1, C2) at line l2. In Fig. 2, the rule assoc_add(xsrc, ysrc, asrc, 1, 2) is added at line 20. [Line A9]: Enable the custom automation function named reduce_maydiff, which tries to find and insert appropriate reduce_maydiff rules when necessary. In Fig. 2, it figures out that reduce_maydiff(y) is needed at line 20.

Automation An automation function works as follows. Given a pair \((Q, Q')\) of ERHL assertions, it examines the assertions Q and Q', and tries to find inference rules I such that the result of applying I to Q trivially implies Q'.

It is important to note that automation functions do not need to be verified (i.e., not a part of TCB) because all they do is inserting inference rules, which is a part of proof construction, not that of proof checking.

3 Register Promotion

Register promotion optimization, the mem2reg pass of LLVM, transforms memory accesses to locally allocated memory locations into register accesses, provided that the memory location is only used for loads and stores (i.e., never copied or escaped). This translation is important because register accesses are cheaper than memory accesses, and are subject to further optimizations.

The optimization also performs the SSA transformation so that the target program has the SSA property. This transformation is necessary because there can be statically multiple stores to a single location, and just transforming them to writes to a single register would break the SSA property.

In this section, we show how we generate and validate proofs for the mem2reg optimization.

3.1 Translation Example

The shaded part of Fig. 3 shows an example translation of the mem2reg optimization, where all memory accesses via \text{p} is promoted to register accesses to \text{p} and uses of 42 and x. Note that c, x, and q are the function parameters.

More specifically, the allocation, load and store instructions to \text{p} are removed (ignore \text{loop} for now), and every use of the result of a load from \text{p} is replaced by the value stored in \*\text{p} at the time of the load. For example, in Fig. 3, the compiler figures out that \*\text{p} contains 42 at line 20 (and so does the register a) due to the store of 42 in \*\text{p} at line 11, and thus replaces the use of a with 42 at line 21. This translation is sound because (i) the assertion \*\text{p}

\text{src} = 42 holds from line 11

\[9\]The instruction that defines x is unique thanks to the SSA property.
We show how to turn the intuition for soundness into a for-
inserted at lines 10, 11, 20, 30, 40 because the instructions
features of ERHL by explaining each part of the proof.

3.2 ERHL Proof

We show how to turn the intuition for soundness into a for-
control flow, the compiler inserts a phinode, which is a unique
constuct in the SSA form and assigns different values to a
register depending on the control flow. For example, at line
40, *p contains 42 if the control comes from B_{left}, and x if it
comes from B_{right}. In this case, the compiler inserts a phin-
od e p1 := \phi(42, x) at the beginning of B_{exit}, which defines
p1 to be 42 when coming from B_{left} and x when coming from B_{right}. Then, the use of the register b containing the
loaded value from *p can be replaced by p1 at line 41.

Logical No-Ops for Instruction Alignment

Logical NoOps, denoted 1nop, can be inserted as a part of proof in order
to align source and target instructions when their alignment is
bro-ken by a translation. For example, in Fig. 3, 1nop is
inserted at lines 10, 11, 20, 30, 40 because the instructions
there are removed by mem2reg.

Note that 1nop is logical because it is absent from the
real IR code and used only for validation purposes. During
validation, it is interpreted as doing nothing (i.e., no-op).

Ghost Registers for Relational Assertions

For complex optimizations, we often need to state relational properties
(i.e., relating the source and target states) in a proof. For example, in Fig. 3, we need to state *p_{src} = p1_{tgt} before line
40, which relates a value in the source (*p_{src}) with that in the
target (p1_{tgt}).

Though not directly supported in ERHL, such relational
properties can be encoded using ghost registers. Specifically, we can encode e_{src} = e'_{tgt} using a fresh ghost register \hat{g}:

\{ e_{src} = \hat{g}_{src}, \hat{g}_{tgt} = e'_{tgt}, MD(M) \} with \hat{g} \notin M

Since the ghost register \hat{g} is not in the maydiff set, we have
\hat{g}_{src} = \hat{g}_{tgt}, which, by transitivity, implies e_{src} = e'_{tgt}. For example, in Fig. 3, the assertion \{ *p_{src} = \hat{p}_{src}, \hat{p}_{tgt} = p1_{tgt},
MD(p1, a, b) \} before line 40 effectively states *p_{src} = p1_{tgt}.

Note that the ghost register \hat{p} has nothing to do with the
physical register p and we use (\sim) for ghost registers to
distinguish them from the physical ones.

Ghost registers are logical ones that do not exist in physical
program states. Instead, they are existentially quantified
in the semantics of ERHL assertions. More specifically, a
pair of source and target states (\sigma_{src}, \sigma_{tgt}) satisfies an ERHL
assertion P, if there exists a pair of source and target ghost
register files (r_{src}, r_{tgt}) such that the pair of \sigma_{src} extended
with r_{src} and \sigma_{tgt} extended with r_{tgt} satisfies P.

Taking ghost registers into account, the proof in Fig. 3
has five relational assertions: *p_{src} = 42_{tgt} between line 11
and the end of B_{left}, a_{src} = 42_{tgt} between line 20 and line 21,
*p_{src} = x_{tgt} between line 30 and the end of B_{right}, *p_{src} = p1_{tgt}
between the beginning of B_{exit} and line 41, and b_{src} = p1_{tgt}
between line 40 and line 41. It is easy to see that these asser-
tions correctly capture the relational properties caused by
executing different instructions in the source and target.

Uniqueness Predicate for Isolation

We can use the predicate Uniq in order to state that an address is completely
isolated. For example, in Fig. 3, we have Uniq(p_{src}) at every
line. It means that in the source, if p contains an address \ell, (i)
\ell is not aliased with any address stored in the other registers
and the memory (i.e., they point to disjoint memory blocks); and (ii) \ell is private (i.e., it is not in the public part of the
memory injection) meaning that it has no corresponding
equivalent address in the target. In other words, the address
contained in p should point to a completely isolated block.

Note that ERHL also supports memory predicates weaker than
Uniq(p): (i) the privateness predicate, Priv(p), which states that the address in p is private; and (ii) the noalias
predicate, p \perp q, which states that the addresses in p and q
point to disjoint memory blocks.
Maydiff Sets Finally, we have MD({p, p1, a, b}) at every line because these registers are removed or introduced so that they have different values in the source and target.

3.3 Proof Validation

We show how our proof checker validates the ERHL proof.

Entry The proof checker checks that the entry assertion, ([Uniq(psrc), MD({p, p1, a, b})]), holds for initial states. It accepts the assertion Uniq(pν) since p is a local register and thus contains the undef value initially, which is not an address. It also accepts every maydiff set since the source and target registers initially contain equivalent values.

Allocation of the Promoted Location At line 10, the proof checker allocates an allocation, p := alloc(), in the source and lnop in the target. In this case, it computes a post-assertion from the pre-assertion by (i) removing all assertions containing psrc because psrc is updated, (ii) adding {Uniq(psrc), *psrc = undef} because p contains a newly allocated address, and then (iii) adding p to the maydiff set. Thus, we have {Uniq(psrc), *psrc = undef, MD({p, p1, a, b})}, from which the assertion after line 10 trivially follows.

Stores to the Promoted Location At line 30 (and similarly at line 11), the proof checker allows a store, *p := x, in the source and lnop in the target because *psrc is private (i.e., has no corresponding target address) due to Uniq(psrc) in the pre-assertion. In this case, it computes a post-assertion by (i) removing all and only the assertions containing *psrc because *psrc is updated and psrc has no alias with any other address due to Uniq(psrc), and then (ii) adding {*psrc = xsrc}. Thus, we have {Uniq(psrc), *psrc = xsrc, MD({p, p1, a, b})}.

At this point, the proof gives the rule intro_ghost(p, x), which first makes p fresh by removing all assertions about p and removing p from the maydiff set and then adds {xsrc = psrc, ptgt = xsrc} when x is not in the maydiff set. Thus, we have {Uniq(psrc), *psrc = xsrc, xsrc = psrc, ptgt = xsrc, MD({p, p1, a, b})}. Then, the proof gives the rule transitivity(*psrc, xsrc, psrc), which derives *psrc = psrc from *psrc = xsrc and xsrc = psrc. Then the assertion after line 30 trivially follows. (See Appendix I for the definitions of intro_ghost and transitivity.)

Phinodes At the phinode of Bexit, the proof checker validates the assertion separately for each incoming block. For the incoming block Bleft, the proof checker computes a post-assertion by (i) removing all assertions containing p1tgt because p1tgt is updated, (ii) adding 42 = p1tgt because p1 := 42 is executed in the target when control comes from Bleft, and then (iii) adding p1 to the maydiff set. Then the proof gives the inference rule transitivity(p1tgt, 42, p1tgt), which derives p1tgt = p1tgt from which the assertion after the phinode follows trivially. For the incoming block Bright, validation succeeds similarly, where the proof gives the inference rule transitivity(p1tgt, x1tgt, p1tgt).

Note that for presentation purposes here we simplified the post-assertion computation for phinodes. ERHL performs a more general version to handle cyclic control flows (see §4).

Loads from the Promoted Location At line 40 (and similarly at line 20), the proof checker allows a load, b := *p, in the source and lnop in the target. In this case, it computes a post-assertion by (i) removing all assertions containing bsrc because bsrc is updated, (ii) adding bsrc = *psrc and then (iii) adding b to the maydiff set. Thus, we have {Uniq(psrc), *psrc = bsrc, btgt = p1tgt, bsrc = *psrc, MD({p, p1, a, b})}.

At this point, the proof gives the rule intro_ghost(b, p), which adds {bsrc = bsrc, btgt = btgt} because p is not in the maydiff set. Then the proof gives appropriate transitivity rules, which derives bsrc = *psrc = bsrc and btgt = btgt = p1tgt, from which the assertion after line 40 trivially follows.

Equivalence Checking At lines 21, 31 and 41, the proof checker checks that the behaviors of the source and target instructions are equivalent. Specifically, it checks that equivalent values are passed to the same function (at line 21) and stored at equivalent public locations (at lines 31, 41) because these can be observed by other functions. These checks succeed thanks to the relational assertions ({asrc = asrc, atgt = 42} at line 21, {bsrc = bsrc, btgt = btgt} at line 41).

Alias Checking At lines 21, 31, and 41, the proof checker computes post-assertions using memory alias information. In general, for a function call or store instruction, since it updates the public part of the memory, the proof checker removes all assertions about values stored in memory locations p (i.e., those including *p) unless (i) p is in the private part of the memory (i.e., Priv(p) or Uniq(p)), or (ii) p is not aliased with q (i.e., p ⊥ q) in case q is updated by the store instruction. At lines 21, 31 and 41, thanks to Uniq(psrc), the assertions about *psrc are preserved.

Note that in the example of Fig. 3, it suffices to use Priv(psrc) instead of Uniq(psrc). However, in general when more than one locations are promoted, we need to know that those promoted locations are not aliased with each other, which follows from Uniq(psrc) for each promoted location p. Also for the sake of performance, we use Uniq instead of introducing ⊥ between each pair of promoted locations.

3.4 Proof Generation

LLVM’s mem2reg pass consists of three algorithms: the general register promotion algorithm, a specialized algorithm for the case that the promotable location is stored at most once, and another for the case that the location is used only within a single block. In this section we explain the general algorithm and its proof generation code. Note that we also validate the two specialized algorithms in the same way since they are just degenerate cases.

Algorithm 2 shows the general algorithm implemented in LLVM’s mem2reg pass and the proof generation code, given
in the box, that we inserted. Note that we do not modify the existing compiler code at all and only add the proof generation code. In detail, the overall algorithm including proof generation works as follows.

**Promotable Allocation** [Line A1]: We find a promotable allocation p at line l_a. [Line A2]: Then we insert empty phinodes wherever needed\(^\text{10}\) and add them to the maydiff set globally (i.e., at each line). [Line A3]: We also remove the allocation, insert 1nop at that line, and add Uniq(p_{src}) and MD(p) globally. [Line A4]: In addition, we add the rule intro_ghost(p, undef) because the initial value undef in *p may be used by some load from *p (though it does not happen in Fig. 3). In that case, the code at line A13 would introduce (*p_{src} = p_{src}, p_{tgt} = undef) at line l_a, which will be inferred with the help of intro_ghost(p, undef).

For example, in Fig. 3, the empty phinode p1 := φ(−) is inserted in B_{exit} and p1 is added to the maydiff set globally; and then the allocation at line 10 is removed, 1nop is inserted, Uniq(p_{src}) is added and p is added to the maydiff set globally; and finally intro_ghost(p, undef) is added at line 10.

**Block Traversal** [Lines A5-A7]: We traverse the blocks in DFS order starting from the entry block using the worklist WL. An element of WL consists of triple (B,v,l), where B is the block to visit, v is the value in *p at the beginning of B, and l is the line number where the value v is stored in *p.

[Line A5]: Initially, we put (Entry(F), undef, l_a) in WL and mark the entry block Entry(F) as visited. [Lines A6-A7]: Then we process the blocks in WL one by one. For example, in Fig. 3, B_{entry}, B_{left}, B_{exit}, and B_{right} are visited in order.

**Instruction Traversal** [Line A8]: Given a work (B,v,l), we traverse each instruction (l_i : i) in the block B as follows.

**Store Instructions** [Lines A9-A11]: If i is a store instruction *p := w (line A9), then we remove the instruction (line A10) and update v with the stored value w (line A11). The proof-generation code inserts 1nop, adds intro_ghost(p, w) (line A10) and updates l with the store location l_i (line A11). For example, in Fig. 3, when i is 11: *p := 42, the store i is replaced by 1nop; intro_ghost(p, 42) is added at line 11; and v and l are updated to be 42 and line 11.

**Load Instructions** [Lines A12-A18]: If i is a load instruction x := *p (line A12), then we replace all the uses of x with the current value v (lines A14-A16), and remove the load instruction (line A17). The proof-generation code adds the relational assertion *p_{src} = v_{tgt} from the store site l to the load site l_i (line A13). Then it adds x_{src} = v_{tgt} from the load site l_i to every use site l_j (line A15). It also inserts 1nop, adds x to the maydiff set and adds the rule intro_ghost(x, p) (lines A17-A18).

For example, in Fig. 3, when i is 20: a := *p, the load i is replaced by 1nop; the use of a is replaced by the current value 42 at line 21; *p_{src} = v_{tgt} is added from 11 to 20; a_{src} = 42_{tgt} is added from 20 to 21; a is added to the maydiff set globally; and intro_ghost(a, p) is added at line 20.

**Successors** [Lines A21-A28]: Now we handle the successor (i.e., outgoing) blocks of the current block B. [Line A21]: We traverse each successor block B′ as follows.

- If B′ has a phinode (z := φ(···)) that is inserted by the code at line A2 (line A22), then we update the phinode z’s component for the incoming block B with the value v of *p at the end of B (line A23), and if B′ has not been visited yet, add (B′, z, Begin(B′)) to the worklist WL (line A24). Since the value v is used at the phinode z, we add *p_{src} = v_{tgt} from store location l to the end of B (line A23).

---

\(^{10}\)The optimization uses the “dominance frontier” algorithm \([10]\) in order to list up the blocks that require a phinode. We omit the details for brevity.
For example, in Fig. 3, when \((B, B')\) is \((B_\text{left}, B_\text{exit})\), the phinode \(p_1 := \phi(\neg, \neg)\) is updated to \(p_1 := \phi(42, \neg)\) and \((B_\text{exit}, p_1, \text{Begin}(B_\text{exit}))\) is added to the worklist \(WL\). Also, the \(*p_{\text{src}} = 42_{\text{tgt}}*\) is added from line 11 to the end of \(B_\text{left}\).

- If \(B'\) has no such phinode (line A25), then we simply add \((B', v, l)\) to the worklist \(WL\) if \(B'\) has not been visited yet (line A26). For example, when \((B, B')\) is \((B_\text{entry}, B_\text{right})\), the triple \((B_\text{right}, 42, \text{line 11})\) is added to the worklist.

**[Line A28]**: Finally the successor \(B'\) is marked as visited.

**Inference Rules** As shown in §3.3, the complete proof for mem2reg contains two inference rules, \textit{intro}\_\textit{ghost} and transitivity. The \textit{intro}\_\textit{ghost} rules are explicitly added by the proof generation code shown in Algorithm 2, while the transitivity rules are automatically added by the automation function \textit{transitivity} (line A32).

## 4 Reasoning about Cyclic Data Dependence

In this section, we show how to reason about cyclic data dependence via phinodes.

**Fold-Phi Optimization** Consider the translation below performed by the fold-\phi optimization of \textit{instcombine}, and its ERHL proof. This translation is performed only when \(x\) and \(y\) are no longer used in the target so that they can be optimized away by subsequent optimizations.

\[
B_1: \begin{cases}
\{ x := a + 1 \} & \text{MD}(t) \\
\{ x_{\text{src}} := a_{\text{src}} + 1 \} & \text{MD}(t)
\end{cases}
\]

\[
B_2: \begin{cases}
z := \phi(x, y) & \text{MD}(t, z) \\
w := \phi(42, z) & \text{MD}(t, w)
\end{cases}
\]

\[
\begin{array}{l}
\{ z_{\text{src}} = \hat{z}_{\text{src}} \} \\
\{ \hat{z}_{\text{tgt}} = t_{\text{tgt}} + 1 \} & \text{MD}(t, z) \\
\text{1nop} \sim z := t + 1 & \text{MD}(t) \\
\{ y := z + 1 \} & \text{MD}(t) \\
\{ y_{\text{src}} = z_{\text{src}} + 1 \} & \text{MD}(t)
\end{array}
\]

Note that a set of phinodes can appear at the beginning of a block and are executed simultaneously. For example, in the source program above, when control flows from \(B_2\) to itself, the phinodes \(z\) and \(w\) are set to the old values of \(y\) and \(z\) before executing the phinodes, respectively. Also note that there is a cyclic dependence between \(z\) and \(y\).

The key point of the above ERHL proof is that we temporarily lose the equivalence of \(z_{\text{src}}\) and \(z_{\text{tgt}}\) after the phinodes of \(B_2\) and recover it after line 20. The proof makes this point in two steps. It first shows that \(z_{\text{src}} = t_{\text{tgt}} + 1\) holds after the phinodes, and then using it, shows that \(z_{\text{src}} = z_{\text{tgt}}\) holds after line 20. The second step holds trivially and the first step holds as follows. When control comes from \(B_1\), after the phinodes, we have \(z_{\text{src}} = \hat{x}_{\text{src}} = \hat{a}_{\text{src}} + 1 = \hat{a}_{\text{tgt}} + 1 = t_{\text{tgt}} + 1\), where \(r_{\text{src}}\) denotes the current value of the register \(r\) after the phinodes and \(r_{\text{src}}\) the old value of \(r\) before the phinodes in the source program; and similarly for \(t_{\text{tgt}}\) and \(t_{\text{tgt}}\). When control comes from \(B_2\) after the phinodes, we have \(z_{\text{src}} = \hat{y}_{\text{src}} = \hat{z}_{\text{src}} + 1 = \hat{z}_{\text{tgt}} + 1 = t_{\text{tgt}} + 1\).

**Proof Validation** We show how the ERHL proof checker formally validates the above proof at the most interesting case: the phinodes of \(B_2\) when control comes from itself. First, it computes a post-assertion from the pre-assertion \(\{ y_{\text{src}} = z_{\text{src}} + 1, \text{MD}(t) \} \) as follows.

1. It completely removes all old registers from the pre-assertion and copies assertions about current registers into those about old ones. Thus we have \(\{ y_{\text{src}} = z_{\text{src}} + 1, \hat{y}_{\text{src}} = z_{\text{src}} + 1, \text{MD}(t, \hat{t}) \}\).

2. It computes a post-condition from this new assertion as if the assignments \(z := \hat{y}, w := z\) are executed in the source and \(t := \hat{t}, w := z\) in the target. Specifically, it (i) removes source assertions about \(z, w\) and target ones about \(t, w\) because those registers are updated; (ii) adds \(t, z\) to the maydiff set because they are updated differently in the source and target (note that \(w\) is updated equivalently since \(z\) is not in the maydiff set); and (iii) adds the equalities corresponding to the executed assignments. Thus we have \(\{ \hat{y}_{\text{src}} = \hat{z}_{\text{src}} + 1, \hat{y}_{\text{src}} = \hat{z}_{\text{src}} + 1, \hat{y}_{\text{src}} = \hat{w}_{\text{src}} = \hat{z}_{\text{src}}, \hat{t}_{\text{src}} = \hat{t}_{\text{tgt}}, \text{MD}(t, z) \}\).

Then the proof gives the rule \textit{intro}\_\textit{ghost}(\(z, \hat{z} + 1\)), which adds \(\{ z_{\text{src}} + 1 = \hat{z}_{\text{src}}, \hat{z}_{\text{src}} = z_{\text{src}} + 1 \}\) by transitivity: \(z_{\text{src}} = \hat{y}_{\text{src}} = \hat{z}_{\text{src}} + 1 = \hat{z}_{\text{src}}\) and \(\hat{z}_{\text{tgt}} = z_{\text{tgt}} + 1 = t_{\text{tgt}} + 1\). Finally, all the old registers are completely removed and the assertion after the phinodes \(\{ z_{\text{src}} = \hat{z}_{\text{src}}, \hat{z}_{\text{tgt}} = t_{\text{tgt}} + 1, \text{MD}(t, z) \}\) trivially follows.

## 5 ERHL Proof Checker and Logic

In this section, we explain the proof checker in terms of the ERHL logic, and describe the soundness of the proof checker using the semantic interpretation of the logic. All our results are formally verified in Coq (see Appendix H for details).

**Proof Rules** The checker is based on the proof rules presented in Fig. 4. The checker is given the source and target programs \(Prg_{\text{src}}, Prg_{\text{tgt}}\) and a translation proof \(\Phi\), and tries to deduce \(Prg_{\text{src}} \sim Prg_{\text{tgt}}\) using the \(\text{Sym}\) rule. Here, \(Entry(F)\) denotes the entry block of the function \(F; Prg[F][\xi][B, i]\)
Here, $\text{FinIt}(\sigma)$ means $\sigma$ is a possible initial state of a function call, $[P]$ is the semantic interpretation of the assertion $P$ (see Appendix G for details), and $\alpha$ is a CompCert-style memory injection [19], which basically maps a memory block in the source to an equivalent one in the target.

Second, we give the semantic interpretation of the Hoare triple for non-call instructions $I_{src}, I_{tgt}$ as a simulation step:

$$\left[ \{P\} I_{src} \sim I_{tgt} \{Q\} \right] \overset{\text{def}}{=} \forall \sigma_{src}, \sigma_{tgt}, \alpha. \left[ P \right]_{\alpha}^{*} \{\sigma_{src}, \sigma_{tgt}\} \implies \exists \sigma'_{src}, \sigma'_{tgt}, \alpha'. \left[ Q \right]_{\alpha'}^{*} \{\sigma'_{src}, \sigma'_{tgt}\} \land \alpha \subsetneq \alpha'$$

where, $\text{Intr}(\sigma)$ is the next instruction to execute in the program state $\sigma$, and $\sigma \rightarrow \sigma'$ means the state $\sigma$ steps to $\sigma'$ emitting an observable event $\sigma$. Also, $\subsetneq$ is the extension relation of memory injection.

For call instructions $I_{src}, I_{tgt}$, $\left[ \{P\} I_{src} \sim I_{tgt} \{Q\} \right]$ basically states that $Q$ is satisfied by all possible equivalent returns states when an arbitrary function call is called from states satisfying $P$ (see Appendix H for details). We followed the basic approach of parametric bisimulation [13].

The semantic interpretation of $\Rightarrow$ is as follows:

$$\left[ Q \Rightarrow Q' \right] \overset{\text{def}}{=} \forall \sigma_{src}, \alpha. \left[ Q \right]_{\alpha}^{*} \{\sigma_{src}, \sigma_{tgt}\} \implies \exists \sigma'_{src}, \alpha'. \left[ Q' \right]_{\alpha'}^{*} \{\sigma'_{src}, \sigma'_{tgt}\} \land \alpha \subsetneq \alpha'$$

For the soundness of (APPLYINF), every custom rule should satisfy that $\left[ Q \Rightarrow \text{ApplyInRule}(\text{rule}, Q) \right]$ holds for all $Q$.

6 Implementation

We developed the Crellvm framework for LLVM 3.7.1.

Proof-Generation Code We wrote proof-generation codes for register promotion (mem2reg) and GVN-PRE (gvn). To demonstrate the generality of ERHL logic and proof checker, we also wrote proof-generation codes for a part of loop invariant code motion (licm) and 139 micro-optimizations of instruction combining (instcombine, see Appendix D for the list of the supported micro-optimizations).

We explicitly mark as "not supported" for translations using operations not supported by Vellvm, or relying on deep analyses such as division-by-zero and alias analyses. Fig. 5 shows the SLOC in C++ of the compiler and proof generation code for each pass. The SLOC ratio of the proof generation code to that of the corresponding compiler code is 37.5% for mem2reg, 40.2% for gvn, 40.5% for licm, and 193.3% for instcombine. The Crellvm infrastructure for proof generation consists of 1,708 lines for common library and 15,980 lines for JSON serialization library, of which 72.2% is automatically generated from 2,079 SLOC in a simple DSL.

Inference Rules In the proof checker we installed 221 custom inference rules, of which 202 are arithmetic rules like assoc_add. All 9 non-arithmetic rules used for mem2reg, gvn,
When proof checking fails, it tells us a logical reason for it.

The proof checker is 2,987 SLOC, and its verification is 18,934 (see §5). It is worth noting that we achieved the same level of guarantee as CompCert for the translations that are validated by the proof checker using only verified inference rules.

We used the formal semantics of LLVM IR from the Vellvm project [43], but significantly upgraded the semantics in various ways. In particular, Vellvm used the CompCert memory model [19] version 1.9 and we upgraded it to version 2.4 in order to use the notion of permission in the LLVM semantics; and added the switch instruction to the formalization of LLVM IR. Note that Vellvm has a simpler memory model than the LLVM’s informal official one (e.g., pointer equality tests and pointer-integer casts are more undefined).

In total, our Coq development consists of 25,970 SLOC. The proof checker is 2,987 SLOC, and its verification is 18,934 SLOC. The 221 inference rules are 2,193 SLOC, and the verification of 9 rules took 1,856 SLOC. Note that the underlying semantics model of Vellvm consists of 39,307 SLOC.

Experience Writing proof-generation code was an iterative process: we had to repeat bug-fix processes many times. When proof checking fails, it tells us a logical reason for the failure so that we could easily identify the bug in proof-generation code (or else in the compiler). We believe the iteration could be shortened if we collaborated with LLVM developers.

Custom functions for automatically finding inference rules are greatly helpful for developing proof-generation algorithms. Using such automation, we could develop much simpler proof-generation algorithms for mem2reg and gvn, compared to our initial development, by making the code size less than half and speeding up more than twice.

Crellvm is less cost-effective for peephole optimizations in instcombine. We had to write 1.9 lines of proof-generation code for each line of the corresponding compiler code, and we did not verify arithmetic inference rules. Even though Crellvm achieves higher level of reliability, we think more automated approaches such as Alive [20] is more cost-effective for peephole optimizations.

7 Experiment

Benchmarks Using Crellvm, we validated the compilation of the SPEC CINT2006 C Benchmarks [4], LLVM nightly test suite, and five open-source projects written in C (the biggest benchmarks used in [27][11], totaling 5,3 million LOC in C. We omitted 4 files from the benchmarks because 3 of them contain instructions currently not supported by Vellvm, including the indirectbr instruction, and the other is too big (151MB) for our proof checker to handle.

Fig. 6 summarizes the validation results and the time spent on running the proof-generation codes and the proof checker for each optimization pass. In the experiment, we compiled each benchmark program with the -O2 flag, and validated the intermediate translations with the generated proofs. For more detailed results, see Appendix A.

We show the total number of translation steps (#V), the number of not-supported translations (#NS), and the number of translations failed at validation (#F). The rest of the translations (i.e., #V − #F − #NS) succeeded in validation. Also, all the successful translations were shown to be equivalent to the original translations using the llvm-diff tool. During the experiment, we also found and reported a bug in llvm-diff, which has been confirmed and fixed.

Out of 2,195K validations in total, 1,623K (73.9%) are successfully validated. All 305 (0.01%) failures are due to compiler bugs: 295 are due to the two bugs we found in gvn, and 10 are due to a register promotion bug we found (see §1.2). After fixing the bugs we found, the compiler no longer failed at validation. 572.0K (26.1%) translations are currently not supported in our validator, among which 555.7K (97.1%) use instructions not supported by Vellvm: vector operations 515.0K (92.7%), aggregate type operations 30.4K (5.47%), debug attributes 8,673, (1.56%) and atomic operations 1,714 (0.31%). 13,013 (2.27%) use the alias and division-by-zero instructions; and 676 (0.12%) require deeper analysis on functions such as read-only function analysis.

We measured the time spent on performing each optimization in the original compiler (Orig); on performing each optimization and calculate validation proofs in the modified compiler (PCal); on writing and reading the source and target programs with the proofs via files (I/O); and on validating the proofs by the proof checker (PCheck). We measured the time taken by each job, and simply summed them up to compute the total time taken.

In the experiment, we embarrassingly parallelized compilation and validation jobs and fully utilized the 96 hardware threads from four identical workstations with Intel Xeon E5-2630 CPU (2.6GHz, 12 cores, 2 hardware threads per core),

\[11\]We omitted Linux, since it is currently not compiled with LLVM (see [3]).
128GB RAM, and 1TB SSD (Samsung 850 PRO). The whole experiment took about three hours in wall clock.

**Validating Randomly Generated Programs** We randomly generated 1,000 C programs using Csmith [41], compiled them with -O2 flag, and validated the intermediate translations with the generated proofs. All 50,064 mem2reg validations are successfully validated, except for one due to the mem2reg bug we found (see §1.2). Out of 42,584 GVN-PRE validations, 11,816 (27.7%) are currently not supported, and the other 30,768 (72.3%) are all successfully validated.

**Performance** Proof checking takes much more time than regular compilation, but we believe it is still reasonable for compiler writers to use Crel LLVM for stabilizing compilers. Also, as we have shown in the experiment, compiler writers can further reduce runtime by checking proofs in parallel. Furthermore, there is still a large room for performance improvement as we have not done any serious performance analysis and tuning for the proof checker. In particular, we believe we can significantly reduce I/O time, which is the current bottleneck, by writing proofs in binary format rather than in plain-text JSON format. Currently, the proof size per source file is about 8.2 MB in total (5.5 MB for mem2reg, 0.2 MB for gvn, 2.2 MB for llicm, 0.3 MB for instcombine).

8 Related Work

A large number of prior work on improving reliability of compiler are roughly classified into the following categories.

**Credible Compilation** Rinard et al. [32], who coined the term credible compilation, proposed the framework of credible compilation and presented a relational Hoare logic, in which one can reason about register allocation and instruction scheduling optimizations in the presence of pointer aliasing. Independently, Benton [8] proposed a relational Hoare logic for a functional language. However, their logics are designed for simple languages, and the framework has not been implemented and applied to compilers.

Namjoshi et al. [23, 24] presented a “proof of concept” implementation of credible compilation (or a witnessing compiler in their terminology) for LLVM optimizations such as constant propagation, deadcode elimination, and LICM. However, the work can be seen as rather preliminary for the following reasons. First, their proof checker supports a small subset of LLVM IR, most notably ignoring memory operations. Second, it assumes that main functions of the compiler are correct. For example, it assumes that the constant folding function of LLVM is correct.

Verified translation validation is similar to verified credible compilation but differs in that it develops a verified validator specialized for a particular optimization, rather than developing a proof checker for a general logic. Various verified translation validators have been developed for CompCert: instruction scheduling [38], lazy code motion [39], software pipelining [40]; register allocation [31]; SSA transformation [7]; and GVN and sparse conditional constant propagation (SCCP) [11].

(Foundational) proof carrying code (PCC) [6, 25] is similar to (verified) credible compilation, but it employs a (verified) unary logic for validating safety properties of the generated target program.

**Translation Validation** Translation validators check equivalence (precisely, refinement) between any two given source and target programs. Since they are agnostic to the compiler’s internal logic and regard the target optimization as a black box, they are inherently incomplete. Therefore, a variety of translation validators with different heuristics and trade-offs were proposed [12, 26, 28, 29, 33–35, 37, 42, 45, 46]. In particular, Tristan et al. [37] and Stepp et al. [34] developed translation validators for LLVM optimization passes, including dead code elimination, GVN-PRE, constant propagation, and LICM. However, they failed at about 20% of the validations, most of which are likely to be false alarms.

**Compiler Verification** Verified compilers provide the highest level of reliability by proving the semantics preservation property for all input programs in a proof assistant. CompCert [17, 18] is the most sophisticated formally verified optimizing C compiler, whose correctness is proved in Coq [1], and CakeML [15] is an optimizing ML compiler formally verified in the HOL4 theorem prover [2]. However, verifying a full-fledged compiler is highly costly and verified compilers are usually much less performant than production compilers.

Zhao et al. [43, 44] implemented and verified the vmem2reg pass for LLVM in Coq, but its algorithm is significantly simplified compared to that in LLVM. Their simplified algorithm is based on a rewriting logic in which each rewriting step preserves semantics and each intermediate program is type-checked. On the other hand, LLVM’s register promotion algorithm temporarily breaks semantics preservation property and even the intermediate programs are not type-checked, because ill-formed empty phi nodes are inserted in the middle and their arguments are filled later. According to the authors, this renders the formal verification hard for the register promotion implementation in LLVM.

**DSL for Optimizations** Lopes et al. [20–22] presented Alive, a DSL for writing peephole optimizations using the SMT solver Z3 [5]. With Alive, one can either prove the correctness of an optimization or find a counterexample. They ported 300 micro-optimizations of instcombine to Alive, and in doing so they found 8 bugs in instcombine. However, Alive is limited to supporting only peephole optimizations that do not involve reasoning about cyclic control flows. In addition, Alive makes simplifying assumptions on the LLVM semantics, and their encoding of an optimization into SMT queries is a part of TCB. Tatlock and Lerner [36] also presented a DSL for writing CompCert optimizations based on
a rewriting logic, but it is not general enough to support GVN-PRE and register promotion.

**Compiler Testing** Random testing tools such as CSmith [9, 30, 41] and EMI [16] have been very successful. They have found hundreds of bugs in GCC and LLVM. However, most of them are found in the `instcombine` pass and none of them in `mem2reg` and `gvn`.

### 9 Conclusion

We have demonstrated that the credible compilation approach scales to the production compiler LLVM by developing our Crellvm framework. We also empirically show that Crellvm can be an effective tool for achieving high reliability of major optimizations by discovering four long-standing bugs in the `mem2reg` and `gvn` passes, for which to the best of our knowledge, no previous systematic approaches have found bugs.
References


Crellvm: Verified Credible Compilation for LLVM

PDL’18, June 18–22, 2018, Philadelphia, PA, USA

LOC Result
mem2reg gvn licm instcombine
#V #F #NS #V #F #NS #V #F #NS #V #F #NS

400.perlbench 168.16K 1.75K 0 1 11.57K 17 0 2.39K 0 105 59.34K 0 6.94K
401.bzip2 8.29K 90 0 0 1.57K 0 0 443 0 36 4.52K 0 1.84K
403.gcc 517.52K 5.26K 0 5 36.01K 20 0 8.34K 0 1.10K 140.26K 0 4.87K
429.mcf 2.69K 24 0 0 144 0 0 29 0 2 487 0 53
433.milc 15.04K 235 0 2 2.03K 0 0 1.78K 0 311 3.69K 0 471
445.gobmk 196.24K 2.64K 0 2 6.95K 0 0 2.54K 0 179 11.06K 0 3.41K
456.hmmer 35.99K 558 0 0 3.40K 3 2 2.54K 0 179 11.06K 0 3.41K
458.sjeng 13.85K 130 0 0 1.73K 0 0 355 0 69 3.78K 0 224
462.libquantum 4.36K 123 0 79 0 0 348 0 258 0 19.06K 0 10.4K
464.h264ref 51.58K 2.64K 0 1 6.95K 0 0 2.54K 0 179 11.06K 0 3.41K
470.lbm 1.16K 19 0 0 77 0 0 61 0 2 174 0 51
482.sphinx3 25.09K 364 0 0 1.60K 0 2 105 0 120 6.24K 0 1.04K
sendmail-8.15.2 138.68K 536 0 403 4.54K 0 107 1.81K 0 163 14.12K 0 396
emacs-25.1 463.54K 5.15K 0 4 28.10K 14 25 7.76K 0 622 112.94K 0 9.42K
python-3.4.1 486.38K 8.78K 0 89 27.37K 117 26 9.13K 0 163 14.12K 0 396
gimp-2.8.18 797.65K 13.00K 0 9.18K 66.24K 26 6.82K 37.49K 0 7.79K 246.21K 0 82.75K
LLVM nightly test 1358.76K 17.98K 3 291 115.79K 49 240 59.04K 0 7.79K 711.14K 0 352.46K
Total 5289.18K 76.62K 10 10.58K 357.17K 295 7.79K 168.20K 0 24.93K 1593.44K 0 528.75K

Figure 7. Validation Results

Fig. 7 and Fig. 8 shows the validation results and the time spent on running the proof-generation codes and the proof checker for each benchmark program and optimization pass.

Miscompilation of a Realistic Program due to a Mem2reg Bug

One of the bug we found in mem2reg miscompiles the following program:

```
#include <stdio.h>

int sqr(int i, int prev, int cur) {
    return cur * cur;
}

int diffsqr(int i, int prev, int cur) {
    if (i==0) return 0; else return (cur-prev) * (cur-prev);
}
```

Figure 8. Time Spent on Running the Proof-Generation Codes and the Proof Checker

A Experimental Results

Fig. 7 and Fig. 8 shows the validation results and the time spent on running the proof-generation codes and the proof checker for each benchmark program and optimization pass.

B Miscompilation of a Realistic Program due to a Mem2reg Bug

One of the bug we found in mem2reg miscompiles the following program:

```
#include <stdio.h>

int sqr(int i, int prev, int cur) {
    return cur * cur;
}

int diffsqr(int i, int prev, int cur) {
    if (i==0) return 0; else return (cur-prev) * (cur-prev);
}
```
We use a PRE translation example because it is more interesting than a GVN example. The with more than one value for brevity.

```c
void foo(int arr[], int n) {
    int sqrsq = 0, diffsqrsq = 0;
    int i, prev, cur;
    for (i = 0; i < n; ++i) {
        prev = cur; cur = arr[i];
        sqrsq += sqr(i, prev, cur);
        diffsqrsq += diffsqr(i, prev, cur);
    }
    printf ("square sum=%d, diff sqa sum=%d \n", sqrsq, diffsqrsq);
}

int main () {
    int a[3] = {1, 2, 5};
    foo(a, 3);
    return 0;
}
```

The function `foo()` takes an array `a` and its size `n` and prints the sum of the squares of the numbers in `a`, and the sum of the squares of the differences of adjacent numbers in `a`, i.e.,

\[
\sum_{i=0}^{n-1} a[i]^2 \quad \text{and} \quad \sum_{i=1}^{n-1} (a[i] - a[i-1])^2.
\]

The function `sqr()` calculates the two summations in a single loop. Note that the function `difsqr(i, prev, cur)` returns `(cur - prev)^2` if `i > 0`, and zero otherwise.

Clang 3.7.1 with `-O2` flag miscompiles this program. The compilation result prints out 30 and 0 instead of the correct answer 30 (= \(1^2 + 2^2 + 5^2\)) and 10 (= \((2 - 1)^2 + (5 - 2)^2\)). This is due to the `mem2reg` bug we found on the special case that all stores to a promotable location is in a single block. In essence, the loads from the local variable `prev` are illegally promoted to `undef`, the function call to `difsqr()` is inlined, and exploiting the `undef` semantics, the result of the inlined call is optimized to 0.

Even though the program invokes undefined behavior according to the C standard, we believe it is still likely to be written in the real-world: a programmer may logically conclude that the `undef` value is never used, and the program is safe.

### C Global Value Numbering with Partial Redundancy Elimination

Global Value Numbering optimization (GVN), which is implemented in the `gvn` pass of LLVM, detects and removes redundant instructions. GVN algorithm first groups expressions and values into equivalent classes and assigns a unique "value number" to each class. Then, a leader value is chosen for each class, and all the non-leader, redundant, instructions are substituted with the leader value. The `gvn` pass also does Partial Redundancy Elimination optimization (PRE), which eliminates instructions that are partially redundant depending on the control flow.

In this section, we show how we generate and validate proofs for GVN-PRE optimization. It is worth noting that even though the GVN and PRE algorithms are separately written in the `gvn` pass, their validation logics are so similar that the resemblance of their validation logics enabled us to write a single proof generation code that uniformly works for both GVN and PRE.

#### C.1 Translation Example

We use a PRE translation example because it is more interesting than a GVN example. The shaded part of Fig. 9 shows an example translation of the PRE optimization, where the registers `n` and `c1` are the function parameters.

First, by analyzing the source program, GVN assigns unique value numbers to the classes of equivalent values and expressions. For example, in Fig. 9, GVN constructs the following tables, `VT`, `ET` and `LT`.

\[
VT = [x_1, x_2 \mapsto \ast; \quad y_1, y_2, y_3 \mapsto \circ]
\]

\[
ET = [n - 2 \mapsto \ast; \quad 1 + \ast \mapsto \circ]
\]

\[
LT = [\ldots; \quad B_{\text{empty}} \mapsto [\ast \mapsto x_1; \quad \ast \mapsto y_1; \quad \ast \mapsto 10]; \quad B_{\text{right}} \mapsto [\ast \mapsto x_2; \quad \ast \mapsto y_2]; \quad \ldots]
\]

Originally, The `gvn` pass also assigns value numbers to singleton classes, but in this example, we only consider those classes with more than one value for brevity.

Here, the value table `VT` assigns value number \(\ast\) to \(x_1\) and \(x_2\), and \(\circ\) to \(y_1\), \(y_2\) and \(y_3\); and the expression table `ET` assigns value number \(\ast\) to the expression \(n - 2\), and \(\circ\) to \(1 + \ast\). This means that, at any point of execution, there exist some values for \(\ast\) and \(\circ\) such that \(x_1\), \(x_2\) and \(n - 2\) evaluate to the value \(\ast\) and \(y_1\), \(y_2\), \(y_3\) and \(1 + \ast\) evaluate to the value \(\circ\) whenever
they are well-defined. It is easy to see that indeed this property holds for the source program in Fig. 9. Finally, the leader table \( LT \) determines the leader value for each value number among the values in each block. Note that the leader values can be different for each block and not every block necessarily has a leader value for every value number. For example, in the block \( B_{\text{entry}} \), the leader value for \( \emptyset \) does not exist because none of the values with value number \( \emptyset \) are defined in the block.

In the example, PRE detects partial redundancies of the instruction \( y_3 := x_1 + 1 \) in \( B_{\text{exit}} \) because \( y_3 \) belongs to \( \emptyset \) and \( B_{\text{exit}} \)'s incoming blocks have a leader value for \( \emptyset \): 10 in \( B_{\text{empty}} \) and \( y_2 \) in \( B_{\text{right}} \). In other words, depending on the control flow, the expression \( y_3 \) is equivalent to either 10 or \( y_2 \). To eliminate the redundant instruction, PRE (i) inserts the phinode \( y_4 := \phi(10,y_2) \) at the beginning of \( B_{\text{exit}} \); (ii) replaces all uses of \( y_3 \) with \( y_4 \) at line 41; and (iii) eliminates \( y_3 := x_1 + 1 \) at line 40. This translation is beneficial because the inserted phinode is compiled down to a move instruction, which is more cost-effective than the eliminated addition instruction.

\[\text{Figure 9. A PRE Example}\]

C.2 ERHL Proof

We can turn our intuition for soundness into a formal proof. The unshaded part of Fig. 9 shows the proof, each assertion of which can be split into three parts.

**Expression Assertions** The red equalities of the form \( e_{\text{src}} = \hat{V}_{i_{\text{src}}} \) or \( \hat{v}_{i_{\text{tgt}}} = e_{\text{tgt}} \) relate expressions with their value numbers according to the expression table \( ET \). For example, the assertion after line 21 states that there exist some values \( \hat{v}_1 \) and \( \hat{v}_2 \) (for \( \emptyset \) and \( \emptyset \)) such that \( \hat{v}_1 = n - 2 \) and \( \hat{v}_2 = 1 + \hat{v}_1 \) hold for both the source and target states.

**Value Assertions** The green equalities of the form \( x_{\text{src}} = \hat{V}_{i_{\text{src}}} \) or \( \hat{v}_{i_{\text{tgt}}} = x_{\text{tgt}} \) relate values with their value numbers according to the value table \( VT \). For example, the assertion after line 21 also states that \( x_1 \) has the value \( \hat{v}_1 \) in the source and \( y_1 \) has the value \( \hat{v}_2 \) in the target.

**Branching Assertions** The blue equalities show properties derived from branching conditions. For example, the equality \( c_{2_{\text{tgt}}} = (y_{1_{\text{tgt}}} == 10) \) at line 21 is derived from the definition of the branching register \( c_2 \), and the equality \( v_{2_{\text{tgt}}} = 10 \) in the block \( B_{\text{empty}} \) is derived from the fact that \( c_2 \) must be true in \( B_{\text{empty}} \) and thus \( y_1 \) is 10 and so is its associated value \( \hat{v}_2 \).
C.3 Proof Validation

The proof in Fig. 9 is validated as follows.

**Adding Value Assertions** At line 40 (and similarly at line 30), the proof checker computes a post-assertion by adding \( y_3^{\text{src}} = x_1^{\text{src}} + 1 \), and deduces, by applying the inference rules from the proof, that:

\[
\begin{align*}
    y_3^{\text{src}} &= x_1^{\text{src}} + 1 \quad \text{(post-assertion)} \\
    &= v_1^{\text{src}} + 1 \quad \text{(substitution)} \\
    &= 1 + v_1^{\text{src}} \quad \text{(commutativity_add)} \\
    &= v_2^{\text{src}} \\
\end{align*}
\]

from which the next assertion trivially follows.

**Adding Expression Assertions** At line 31 (and similarly at lines 10 and 20), the proof checker computes a post-assertion by adding \( y_2^{\text{src}} = x_2^{\text{src}} + 1, x_2^{\text{tgt}} + 1 = y_2^{\text{tgt}} \). Then the proof gives the inference rule intro_ghost(1 + v1, v2), which adds \( 1 + v_1^{\text{src}} = v_2^{\text{src}}, v_2^{\text{tgt}} = 1 + v_1^{\text{tgt}} \). Then the proof checker deduces \( v_2^{\text{tgt}} = y_2^{\text{tgt}} \), similarly as in line 40.

**Adding Phinode Assertions** At the phinode of B_exit, the assertion is separately validated for each incoming block. For the incoming block B_left (and similarly for B_right), the proof checker computes a post-assertion by adding \( 10 = y_4^{\text{tgt}} \) and adding \( y \) to the maydiff set, and derives \( v_2^{\text{tgt}} = 10 = y_4^{\text{tgt}} \) by applying the transitivity rule from the proof.

**Adding Branching Assertions** At line 21, the proof checker computes a post-assertion by adding \( c_2^{\text{src}} = (y_1^{\text{tgt}} == 10), c_2^{\text{tgt}} = (y_1^{\text{tgt}} == 10) \), from which the next assertion trivially follows.

**Using Branching Assertions** At the beginning of B_empty, the proof checker computes a post-assertion by adding \( \text{true} = c_2^{\text{src}}, \text{true} = c_2^{\text{tgt}} \) from the branching condition, and deduces, by applying the inference rules from the proof, that \( \text{true} = c_2^{\text{tgt}} = (y_1^{\text{tgt}} == 10) \) and

\[
\begin{align*}
    \hat{v}_2^{\text{tgt}} &= y_1^{\text{tgt}} \quad \text{(value assertion)} \\
    &= 10 \quad \text{(from true = (y1tgt == 10) by icmp_to_eq(true,y1tgt,10))}
\end{align*}
\]

from which the next assertion trivially follows.

C.4 Proof Generation

Now we explain how we generate proofs for the GVN-PRE optimization.

We first add codes to GVN and PRE algorithms that generate auxiliary data, RET, RPT and BCT. For example, we compute the following for Fig. 9:

\[
\begin{align*}
    \text{RET} &= [x_1, x_2 \mapsto \{ n_{\text{src}} - 2 = \hat{v}_1^{\text{src}}, v_1^{\text{tgt}} = n_{\text{tgt}} - 2 \}; \\
    \text{y1,y2,y3,y4} \mapsto \{ n_{\text{src}} - 2 = \hat{v}_1^{\text{src}}, v_1^{\text{tgt}} = n_{\text{tgt}} - 2, 1 + \hat{v}_1^{\text{src}} = \hat{v}_2^{\text{src}}, v_2^{\text{tgt}} = 1 + \hat{v}_1^{\text{tgt}} \}] \\
    \text{RPT} &= [x_1, x_2 \mapsto (._.;); \ y1 \mapsto (_.;\_); \ y2,y3 \mapsto (._.;); \ y4 \mapsto (2, 2);] \\
    \text{BCT} &= [(10, 2) \mapsto (\{B_{\text{left}}, \text{true}, y1\})].
\end{align*}
\]

The register expression table RET contains sufficient expression assertions for reasoning about each register. For example, RET(x1) contains \( \{ n_{\text{src}} - 2 = \hat{v}_1^{\text{src}}, v_1^{\text{tgt}} = n_{\text{tgt}} - 2 \} \), which is sufficient for reasoning about x1. The register parameter table RPT contains the value numbers of the instruction parameters for each register. For example, RPT[y1] contains (\_., \_.), which means that the second parameter x1 of the instruction 20: y1 := 1 + x1 has the value number \_.. The branching condition table BCT maps pairs of constants/function parameters and their value numbers to their associated branching information. For example, BCT[(10, 2)] contains (B_{\text{left}}, \text{true}, y1), which means that (i) y1 = 10 holds when control flows from B_{\text{left}} with the branching condition being true (i.e., c2 = true), and (ii) y1’s value number is 2. We can easily construct RET, RPT, and BCT following the construction of VT and ET.

Algorithm 3 presents the common proof-generation code that generate assertions for both GVN and PRE, which is given in a rather functional style for presentation purposes. Specifically, ProofGen(F, r, v, VT, RET, RPT, BCT) generates a proof for the replacement of r with v in function F. In essence, the proof generation code adds assertions, starting from those for r_{src} and v_{tgt} at r’s definition point and recursively down to the arguments, using a worklist and auxiliary data RET, RPT, and BCT. The recursion stops when the target value is the only value of its value number. More concretely, the algorithm works for the example of Fig. 9 as follows.
Algorithm 3 ProofGen($F$: Function, $r$: Register, $v$: Value, $VT$, RET, RPT, BCT)

A1: ($l_r: r := \_ := $) := FindDef($F, r$
A2: $WL := ([l_r, r, VT[r], src], (l_r, v, VT[r], tgt))$
A3: while NonEmpty($WL$) do
A4: $((l_0, x, \hat{n}, side) := WL) := WL$
A5: match FindDef($F, x$) with
A6: $|$ Some ($l: x := e$) $\Rightarrow$
A7: $Assn(RET[x], l, l_0), Assn(x_{side} = \hat{n}_{side}, l, l_0)$
A8: for ($t', y, \hat{m}$) in MatchExpr($e, RPT[x]$) do
A9: $WL := (l', y, \hat{m}, side) := WL$
A10: $|$ Some (ConstantOrParameter($C$)) $\Rightarrow$
A11: $match$ FindBranchingCondition($BCT[(C, \hat{n}], l_0, F))$ with
A12: $|$ Some ($v, y, c, l$) $\Rightarrow$
A13: if FindDef($F, c$) $=$ Some ($l_c: c := e$) then $Assn(c_{side} = e_{side}, l_c, l)$ end if
A14: $Inf(icmp_to_eq(vtgt, C), l)$
A15: $Assn(RET[y], l, l_0), Assn(C_{side} = \hat{n}_{side}, l, l_0)$
A16: $WL := (l, y, \hat{n}, side) := WL$
A17: end match
A18: $|$ Some (ConstantOrParameter($C$)) $\Rightarrow$
A19: $match$ FindBranchingCondition($BCT[(C, \hat{n}], l_0, F))$ with
A20: $|$ Some ($v, y, c, l$) $\Rightarrow$
A21: Auto($GVN\_PRE$)

Initialize Worklist The code finds where $r$ is defined (line A1), and add a work for the source and another for the target to the worklist (line A2). A work ($l_0, x, \hat{n}, side$) means that (i) $x$’s value number is $\hat{n}$, and (ii) line $l_0$ needs the assertion $x_{side} = \hat{n}_{side}$ (where side is either src or tgt) and the expression assertions. For the translation in Fig. 9, $x = y$ is defined at line 40 in the source, so the initial works are (40, y3, v2, src) and (40, y4, v2, tgt). The code processes each work ($l_0, x, \hat{n}, side$) as follows (lines A3-A4).

Processing Registers If $x$ is a register defined as $e$ at $l$ (line A6), the code adds the expression assertions $RET[x]$ for $x$ from $l$ to $l_0$, and the value assertion $x = \hat{n}$ at side (line A7). Consider the work (40, y4, v2, tgt), for example. The register $y4$ is defined by the phinode of $B_{exit}$, so $l$ is the phinode of $B_{exit}$ and $e = \phi(10, y2)$. Hence $RET[y4]$ and $v_{tgt} = y_{4tgt}$ are inserted from the phinode of $B_{exit}$ to line 40. For the work (40, y3, v2, src), since $y3$ is defined at line 40, no assertions are inserted.

In order to justify the inserted assertions at line $l$, the code adds sub-works for the values in $e$ (lines A8-A9). Note that MatchExpr($e, RPT[x]$) matches $e$ against the register parameter table in order to know which value, say $y$, should have which value number, say $\hat{m}$, at line $l'$ (line A8). For example, in order to deduce $y_{3src} = v_{2src}$ after line 40, $x_{1src} = v_{1src}$ should hold before the line, so the code adds (40, x1, v1, src) to the worklist; for justifying $v_{tgt} = y_{4src}$ after the phinode of $B_{exit}$, the code adds ([edge from $B_{empty}$ to $B_{exit}$]), 10, v2, tgt) and ([edge from $B_{right}$ to $B_{exit}$], y2, v2, tgt) to the worklist.

Processing Constants and Parameters If $x$ is a constant or parameter $C$ (line A11), the code looks for a branching condition in $BCT[(C, \hat{n})]$, which justifies that $C$ has the value number $\hat{n}$ at $l_0$ (line A12). The result ($v, y, c, l$) means $y = C$ holds at $l$ thanks to the branching condition $c$ being equal to $v$.

The code adds the branching assertion when necessary (line A14), and also adds an $icmp\_to\_eq$ inference rule (line A15). Also, similarly to the case of registers, the code adds expression and branching assertions (line A16) to the proof, and adds sub-works for the value $y$ (line A17).

For example, consider the work ($E_e, 10, v2, tgt$), where $E_e$ is the edge from $B_{empty}$ to $B_{exit}$. The code finds a branching condition in $BCT[(10, v2)]$ which guarantees that $10 = v2$ holds at $E_e$ (line A12). Block $B_{left}$ has such a branch when $c2$ equals to true ($v, y, c, l$) = $(true, y1, c2, E_l)$ at line 13 where $E_l$ is the edge $B_{left}$ to $B_{empty}$). Hence the code adds $c_{tgt} = ($y1$_{tgt} == 10$) from line 21 to $E_l$ (line A14), and adds the inference rule $icmp\_to\_eq(true, y1_{tgt}, 10)$ at $E_l$ (line A15). Then the code adds the expression assertions $RET[y1]$ for $y1$ and the branching assertion $v_{2tgt} = 10$ from $E_l$ to $E_e$ (line A16), and adds the work ($E_l$, $y1$, v2, tgt) to the worklist (line A16).

Automation Function Throughout the proof we use the $GVN\_PRE$ automation function, which adds intro_ghost, commutativity, and substitution in a specific way for $GVN\_PRE$, and adds transitivity and reduce_maydiff in the same way as for assoc_add and mem2reg.
D Micro-Optimizations in instcombine

We validated the following 139 micro-optimizations in instcombine:


Note that we gave these names and they are not officially used in LLVM.

E Program Points between Two Lines

Assertion(P, i1, i2) in the proof generation code means predicate P should be added to the assertions between i1 and i2. More specifically, the proposition P should be added at every program point appearing in a path from i1 to i2 that does not visit i1 but may visit i2 in-between. Since i1 is the source of the proposition P so that we can get P as a post-assertion every time we visit i1, there is no need to add P along a path from i1 to i2. For example, consider the following program.

The marked area between i1 and i2 is where we should add the proposition P.

Thanks to the SSA property [10], we can efficiently calculate the program points between i1 and i2. First, we can assume that i1 dominates i2 (i.e., one should have visited i1 to reach i2 from the entry point), since the proposition P created at i1 should hold at i2. Then a program point i is on a path from i1 to i2 that does not visit i1 in-between if and only if (i) i1 dominates i and (ii) i is reachable from i without visiting i1. We efficiently check the first condition using the dominator tree [10] and the second condition by a backward BFS search from i2.

In the above example, any program point in the marked area is such that i1 dominates it and i2 is reachable from it without visiting i1. For example, i2 is not reachable from the block B4 without visiting i1, and i1 does not dominate the block B5.

Note that this algorithm is not a part of TCB validation may fail but cannot succeed incorrectly due to bugs in this algorithm.

F Lessdef Predicates

LLVM and CompCert have the notion of undef value, which is designated as the result of erroneous operations. The compilers are allowed to replace undef by an arbitrary value. For example, LLVM’s InstCombine performs the following translation, where the register y is replaced by 1, presumably because \( y_{src} = a_{src} - x_{src} = a_{src} - (a_{src} - 1) = 1 \) holds for any integer value of
a_{src}:

\[
x := a - 1; \quad x := a - 1;
\]
\[
y := a - x; \quad \sim y := a - x;
\]
\[
z := y + 1; \quad z := 1 + 1;
\]

However, if \(a_{src}\) is \text{undef}, then we have \(z_{src} = (a_{src} - (a_{src} - 1)) + 1 = \text{undef}\). \text{undef} is propagated in the arithmetic. Thus the equation \(z_{src} = z_{tgt}\) no longer holds, breaking the equality relationship between the source and target values.

In order to reason such optimizations, we use the CompCert-style \text{lessdef} relation [19] throughout this work instead of the equality as assertion predicates. Concretely, \(x\) is less defined than \(y\), denoted \(x \sqsubset y\), if \(x\) is \text{undef} or it equals to \(y\). For example, we have \(y_{src} = a_{src} - x_{src} = a_{src} - (a_{src} - 1) \sqsubset 1\) and thus \(z_{src} = y_{src} + 1 \sqsubset 1 + 1 = z_{tgt}\) regardless of whether \(a = \text{undef}\) or not, which justifies the above translation.

So far we used the equality instead of \text{lessdef} relation for simplicity of presentation. However, all the assertions presented in this paper works even when equalities are replaced by \text{lessdef} relations. Note that the post-assertion generator should be adapted to \text{lessdef} relations so that they introduce both \(x \sqsubset e\) and \(e \sqsubset x\) after \(x := e\).

G Semantic Interpretation of Assertions

The syntax of assertions is as follows:

\[
\begin{align*}
\text{Reg} & \ni r := \cdots \\
\text{Const} & \ni c := \cdots \\
\text{Typ} & \ni \text{typ} := \text{int} | * \text{typ} | \cdots \\
\text{SV al} & \ni v := r | c \\
\text{Tag} & \ni \text{tag} := \text{Phy} | \text{Ghost} | \text{Old} \\
\text{Reg} \times \text{Tag} & \ni rT := (r, \text{tag}) \\
\text{SV al} \times \text{Tag} & \ni vT := (v, \text{tag}) \\
\text{Expr} & \ni e := \text{add} vT vT | \text{load} vT \text{typ} \text{align} | \cdots \\
\text{Pred} & \ni \text{pred} := e \sqsubset e | \text{Uniq}(r) | \text{Priv}(rT) | vT \perp vT \\
\text{MD} & \ni M := \{rT, \cdots\} \\
\text{AssnU} & \ni S, T := \{\text{pred}, \cdots\} \\
\text{Assn} & \ni P, Q := (S, T, M)
\end{align*}
\]

\(\text{Reg}, \text{Const},\) and \(\text{Typ}\) are the types of LLVM registers, constants, and types. A (static) value is either a register or a constant. Registers and values may be tagged (see /coq/def/Exprs.v:106): the tag \text{Phy} means physical registers, \text{Ghost} means regular ghost registers (§3.2), and \text{Old} means old ghost registers (§4). We write \(r\) for \((r, \text{Phy})\), \(\bar{r}\) for \((r, \text{Ghost})\), and \(\bar{r}\) for \((r, \text{Old})\).

An expression is the right-hand side of a side-effect-free LLVM instructions with operand values being tagged (see /coq/def/Exprs.v:366 in the Coq development\(^\text{12}\)). Notice that load is side-effect-free (except for undefined behavior) so there are load expressions, while store is side-effectful and there are no store expressions. Recall that \(e_1 \sqsubset e_2\) means either \(e_1 = e_2\) or \(e_1 = \text{undef}\), and the predicate \(vT_1 \perp vT_2\) means the addresses in \(vT_1\) and \(vT_2\) point to disjoint memory blocks. A unary assertion is a set of predicates (see /coq/def/Hints.v:34), and a maydiff set is a set of tagged registers. An assertion consists of a unary assertion for source, another for target, and a maydiff set (see /coq/def/Hints.v:41).

We use the following semantic domains:

\[
\begin{align*}
\text{Val} & \ni V := \cdots \\
\text{RF} & \ni rs := \{r \mapsto V, \cdots\} \\
\text{State} & \ni \sigma := \cdots \\
\text{StateT} & \ni \sigma T := (\sigma, rs, rs) \\
\text{Meminj} & \ni a := \cdots \\
\end{align*}
\]

\(^{12}\)We submitted the Coq proof scripts as supplementary material.
Let \( Val \) be the set of (dynamic) values, \( RF \) be the set of register files, \( State \) be the set of states, \( StateT \) be the set of extended states, which are tuples of a state, a register file for ghost registers, and another for old ghost registers. \( Meminj \) is the set of CompCert-style memory injections, which basically maps a source memory block to the equivalent target block [19].

The semantics of assertions is as follows:

\[
[r], [e], [v] : State \rightarrow Val \quad \text{def} \quad \cdots
\]

\[
[rT], [vT] : StateT \rightarrow Val \quad \text{def} \quad \cdots
\]

\[
[pred] : 2^{Blk} \rightarrow StateT \rightarrow \mathbb{P}
\]

\[
[e_1 \sqsubseteq e_2](priv, \sigma T) \quad \text{def} \quad [e_1](\sigma T) \sqsubseteq [e_2](\sigma T)
\]

\[
[\text{Uniq}(r)](priv, \sigma T) \quad \text{def} \quad \forall b, b', o, o'. \ [r](\sigma T) = (b, o) \land (b', o') \in \sigma(T \setminus r) \implies b \neq b'
\]

\[
[\text{Priv}(rT)](priv, \sigma T) \quad \text{def} \quad \forall b, o. \ [rT](\sigma T) = (b, o) \implies b \in priv
\]

\[
[vT_1 \perp vT_2](\sigma T) \quad \text{def} \quad \forall b_1, b_2, a, o_1, o_2. \ [vT_1](\sigma T) = (b_1, a_1) \land [vT_2](\sigma T) = (b_2, a_2) \implies b_1 \neq b_2
\]

\[
[M](\alpha, \sigma T_{src}, \sigma T_{tgt}) \quad \text{def} \quad \forall rT \notin M. \ [rT](\sigma T_{src}) \sim_{\alpha} \ [rT](\sigma T_{tgt})
\]

\[
[S](\sigma T_{src}, \sigma T) \quad \text{def} \quad \forall \text{pred} \in S. \ [\text{pred}](priv, \sigma T)
\]

\[
[(S, T, M)](\sigma T_{src}, \sigma T_{tgt}) \quad \text{def} \quad \exists \sigma T_{src}, \sigma T_{tgt}. \ [M](\alpha, \sigma T_{src}, \sigma T_{tgt}) \land \sigma T_{src}.0 = \sigma_{src} \land [\sigma](\sigma_{src}(\alpha), \sigma T_{src}) \land \sigma T_{tgt}.0 = \sigma_{tgt} \land [\sigma](\sigma_{tgt}(\alpha), \sigma T_{tgt})
\]

Registers, constants, static values have obvious semantics that maps a state to a dynamic value. Tagged registers and values map an extended state to a dynamic value. The semantics of \( pred \) is a predicate over a set of blocks, which represents the set of private blocks, and an extended state (see /coq/proof/InvState.v:346). In particular, \([\text{Uniq}(r)](priv, \sigma T)\) means \( r \) is not aliased with any other values in \( rT \) (see /coq/proof/InvState.v:314), and \([\text{Priv}(rT)](priv, \sigma T)\) means if \( rT \) represents a pointer, it is not in \( priv \) (see /coq/proof/InvState.v:332). \([vT_1 \perp vT_2]\) means if two values are pointers, they point to different memory block (\( b \) for block, \( o \) for offset, see /coq/proof/InvState.v:244). The semantics of a maydiff set is that all corresponding values are injected except for those in the maydiff set. The semantics of an assertion is that there exist ghost and old register files of the source and target such that, together with the source and target states, satisfy the semantics of the source and target assertions and the maydiff set (see /coq/proof/InvState.v:430). Here, \( priv_{src}(\alpha) \) is the set of those source blocks that are not mapped to a target block, and \( priv_{tgt}(\alpha) \) is the set of those target blocks that are not mapped from a source block.

H Details of ERHL Proof Checker

H.1 Semantic Interpretation of Hoare triples for call instructions

We give the semantic interpretation of the Hoare triple for call instructions \( I_{src} = (x_{src} := \text{call } f_{src} \text{ args}_{src}) \) and \( I_{tgt} = (x_{tgt} := \text{call } f_{tgt} \text{ args}_{tgt}) \) as follows:

\[
\begin{align*}
\{ P \} & I_{src} \sim I_{tgt} \{ Q \} \quad \text{def} \quad \\
& \forall \sigma_{src}, \sigma_{tgt}, \alpha. \ [P]_{\alpha}(\sigma_{src}, \sigma_{tgt}) \land \text{Instr}(\sigma_{src}) = I_{src} \land \text{Instr}(\sigma_{tgt}) = I_{tgt} \implies \\
& (I_{src} \sim_{\alpha} I_{tgt}) \land (\text{args}_{src} \sim_{\alpha} \text{args}_{tgt}) \land \\
& \forall v_{src}, v_{tgt}, \sigma'_{src} \sim_{\alpha'} \alpha. \ [T]_{\alpha'}(\sigma'_{src}, v_{tgt}) \land v_{src} \sim_{\alpha'} v_{tgt} \land \sigma_{src} \Rightarrow_{\text{tgt}} \sigma'_{src} \land \sigma_{tgt} \Rightarrow_{\text{tgt}} \sigma'_{tgt} \quad \text{def} \quad \\
& \exists \sigma'_{src}, \sigma'_{tgt} \quad \text{def} \quad \\
\end{align*}
\]

where, \( v_{src} \sim_{\alpha'} v_{tgt} \) means \( v_{src} \) is injected into \( v_{tgt} \) via memory injection \( \alpha \), \( T \) is the assertion with no predicates and the full maydiff set (i.e., the values of every register may differ), and \( \sigma \Rightarrow_{\text{tgt}} \sigma' \) means \( \sigma \) is about to call a function, and \( \sigma' \) is a possible return state after the function call for the case that the callee returns \( v \).
Algorithm 4 CheckEquivBeh($P$, $I_{\text{src}}$, $I_{\text{tgt}}$: Command)

1. match $I_{\text{src}}$, $I_{\text{tgt}}$ with
2. // call
3. | (x_{\text{src}} := call $f_{\text{src}}$ args_{\text{src}}), (x_{\text{tgt}} := call $f_{\text{tgt}}$ args_{\text{tgt}}) \Rightarrow \text{return} (f_{\text{src}} \sim_p f_{\text{tgt}}) \land (\text{args}_{\text{src}} \sim_p \text{args}_{\text{tgt}})
4. | (\_ := call \_\_), | \_ (\_ := call \_\_) \Rightarrow \text{return} false
5. // alloca
6. | (p_{\text{src}} := alloca x_{\text{src}}), (p_{\text{tgt}} := alloca x_{\text{tgt}}) \Rightarrow \text{return} x_{\text{src}} \sim_p x_{\text{tgt}}
7. | (p_{\text{src}} := alloca x_{\text{src}}), (\text{Inop}) \Rightarrow \text{return} true
8. | (\_ := alloca \_), | \_ (\_ := alloca \_) \Rightarrow \text{return} false
9. // store
10. | (\text{store} p_{\text{src}} v_{\text{src}}), (\text{store} p_{\text{tgt}} v_{\text{tgt}}) \Rightarrow \text{return} (p_{\text{src}} \sim_p p_{\text{tgt}}) \land (v_{\text{src}} \sim_p v_{\text{tgt}})
11. | (\text{store} p_{\text{src}} v_{\text{src}}), (\text{Inop}) \Rightarrow \text{return} \text{Priv}(p_{\text{src}}) \in P
12. | (\text{store} \_), | \_ (\text{store} \_) \Rightarrow \text{return} false
13. // target is load
14. | (v_{\text{src}} := \text{load} p_{\text{src}}), (v_{\text{tgt}} := \text{load} p_{\text{tgt}}) \Rightarrow \text{return} p_{\text{src}} \sim_p p_{\text{tgt}}
15. | \_ (v_{\text{tgt}} := \text{load} p_{\text{tgt}}) \Rightarrow \text{return} false
16. // target is div
17. | (\_ := \text{div} \_ b_{\text{src}}), (\_ := \text{div} \_ b_{\text{tgt}}) \Rightarrow \text{return} b_{\text{src}} \sim_p b_{\text{tgt}}
18. | \_ (\_ := \text{div} \_ b_{\text{tgt}}) \Rightarrow \text{return} \text{IsNonzero}(P, b_{\text{tgt}})
19. // misc.
20. | \_ \_ \_ \Rightarrow \text{return} true
21. end match

Algorithm 5 CalcPostAssnCmd($P$: Assn, $I_{\text{src}}$, $I_{\text{tgt}}$: Command): Assn

1. $P' := \text{Prune}(P, I_{\text{src}}, I_{\text{tgt}})$
2. $(P''_{\text{src}}, P''_{\text{tgt}}, M'') := \text{AddMemoryPreds}(I_{\text{src}}, I_{\text{tgt}}, P')$
3. $P'''_{\text{src}} := \text{AddLessDefPreds}(I_{\text{src}}, P''_{\text{src}})$
4. $P'''_{\text{tgt}} := \text{AddLessDefPreds}(I_{\text{tgt}}, P''_{\text{tgt}})$
5. return $\text{ReduceMaydiff}(P'''_{\text{src}}, P'''_{\text{tgt}}, M'')$

The semantic interpretation means that the source and the target is about to call equivalent functions with equivalent arguments, and for all future extension $\alpha'$ of the current memory injection $\alpha$, if the return values and return states are related by $\alpha'$, then there exists a future extension $\alpha''$ of $\alpha'$ for which $[Q]$ is satisfied for the return states.

In order to prove a Hoare triple, we need to guarantee that the source and target call equivalent functions with equivalent arguments, and in turn, we can rely on the fact that the callees return equivalent states. Using this semantics interpretation of calls, we proved semantics preservation of programs using the basic approach of parametric bisimulation [13]. See /coq/proof/SimulationLocal.v:164 and /coq/proof/AdequacyLocal.v:243 for more details.

H.2 Post-Assertion Computation for Commands

CheckEquivBeh Algorithm 4 is the CheckEquivBeh algorithm for commands (see /coq/def/Postcond.v:769). Here, $x_{\text{src}} \sim_p y_{\text{tgt}}$ means one of the followings holds: (i) $x = y$ and $x$ is not in the maydiff set; (ii) $(x_{\text{src}} \not\equiv y_{\text{src}}) \in P_{\text{src}}$ and $y$ is not in the maydiff set; or (iii) $x$ is not in the maydiff set and $(x_{\text{tgt}} \not\equiv y_{\text{tgt}}) \in P_{\text{tgt}}$. Basically, this implies $x_{\text{src}} \sim_a y_{\text{tgt}}$ holds for all $\alpha$ that is compatible with $P$. Also, $e_{\text{src}} \sim_p e'_{\text{tgt}}$ means $e$ and $e'$ are of the same expression kind, e.g., they are both add, and for all matching operands $x, x', x_{\text{src}} \sim_p x'_{\text{tgt}}$ holds. IsNonzero() performs an analysis for proving that the value is nonzero. For simplicity, we omit the details.

In essence, if the target instruction may emit an event or invoke undefined behavior, the source instruction should be similar to that. Note that we allow source load instruction with target Inop instruction, which indeed occurs in the validation of mem2reg.
Algorithm 6 Prune($P$: Assn, $I_{src}$, $I_{tgt}$: Command): Assn

1: $(P_{src}, P_{tgt}, M) := P$
2: $P'_{src} := \text{PruneU}(P_{src}, I_{src})$
3: $P'_{tgt} := \text{PruneU}(P_{tgt}, I_{tgt})$
4: $M' := M \cup \{\text{Def}(I_{src}), \text{Def}(I_{tgt})\}$
5: return $(P'_{src}, P'_{tgt}, M')$

CalcPostAssn Algorithm 5 is the post-assertion computation algorithm for commands $I_{src}, I_{tgt}$ (see /coq/def/Postcond.v:1058 for definition and /coq/proof/SoundPostcondCmd.v:309, /coq/proof/SoundPostcondCall.v:254 for the soundness of (PostAssn) for commands). At line 1, it removes predicates that no longer hold after executing the commands (Prune). Then at line 2, it adds Uniq and Priv predicates when necessary (AddMemoryPreds). At lines 3-4, it adds lessdef predicates to unary assertions for both source and target (AddLessdefPreds), and at line 5, finally tries to remove registers from the maydiff set.

Prune Algorithm 6 is the assertion pruning algorithm for commands $I_{src}, I_{tgt}$ (see /coq/def/Postcond.v:355, /coq/def/Postcond.v:386). At lines 2-3, it removes predicates from unary assertions for both source and target (PruneU). Then at line 4, the left-hand sides of $I_{src}, I_{tgt}$ are added to the maydiff set.

PruneU PruneU($S$, $I$) removes the following memory predicates from a unary assertion:
- If $I$ defines a register $r$, remove all predicates on $r$ (see /coq/def/Postcond.v:373).
- If $I$ is a store instruction $*p := v$, remove all lessdef equations on $*q$ for which we cannot prove $p \perp q$. We can prove $p \perp q$ if we have it as a predicate, or $[p \neq q$, either $p$ or $q$ is unique, and the other is physical] holds (see /coq/def/Postcond.v:341).
- If $I$ is a call instruction, remove all lessdef equations on $*q$ unless Priv($q$) holds (see /coq/def/Postcond.v:409).
- Remove Uniq($p$) if $p$ leaked, i.e., copied to another register, used as the instruction’s operand (see /coq/def/Postcond.v:347), or a function is called unless Priv($p$) holds (see /coq/def/Postcond.v:411).

AddMemoryPreds AddMemoryPreds($I_{src}, I_{tgt}, P$) adds the following memory predicates to $P$:
- If $(I_{src}, I_{tgt})$ are allocations ($p_{src} := \text{alloca}(\cdot \cdot)$), ($p_{tgt} := \text{alloca}(\cdot \cdot$)), add Uniq($p_{src}$) to the source assertion. If $p_{src} = p_{tgt}$ then remove $p_{src}$ from the maydiff set (see /coq/def/Postcond.v:893).
- If $I_{src}$ is an allocation ($p_{src} := \text{alloca}(\cdot \cdot)$) and $I_{tgt}$ is Inop, then add Uniq($p_{src}$), Priv($p_{src}$) to the source assertion (see /coq/def/Postcond.v:905).
- If $(I_{src}, I_{tgt})$ are call instructions ($x_{src} := \text{call}(\cdot \cdot)$), ($x_{tgt} := \text{call}(\cdot \cdot$)) and $x_{src} = x_{tgt}$, then remove $x_{src}$ from the maydiff set (see /coq/def/Postcond.v:927).

AddLessdefPreds AddLessdefPreds($I, S$) adds the following lessdef predicates to $S$:
- If $I$ is a side-effect-free operation ($x := \text{op} \text{ arg}$), add ($x \triangleright= \text{op arg}$) and (op arg $\triangleright= x$) (see /coq/def/Postcond.v:936). Note that load is regarded as side-effect-free.
- If $I$ is a store instruction $*p := v$, add $*p \triangleright= v$ (see /coq/def/Postcond.v:939).
- If $I$ is an allocation instruction $*p := \text{alloca}(\cdot \cdot)$, add $*p \triangleright= \text{undef}$ (see /coq/def/Postcond.v:942).

H.3 Post-Assertion Computation for Phinodes

Phinodes do not emit any events so that any pair of phinodes behaves equivalently. Thus, for phinodes, CheckEquivBeh() checks nothing. As described in §4, CalcPostAssn() works for phinodes as follows (see /coq/def/PostcondPhinodes.v:689 for the definition and /coq/proof/SoundPostcondPhinodes.v:960 for the soundness of (PostAssn) for phinodes):
- Remove old registers, and copy predicates on physical registers into those about old ones (see /coq/def/Postcond.v:673).
- Remove predicates on those registers defined in the phinodes (see /coq/def/Postcond.v:674).
- Add predicates $x \triangleright= y, \bar{y} \triangleright= x$ for each assignment $x := y$ that is performed in the phinodes (see /coq/def/Postcond.v:680).
- Tries to reduce the maydiff set (see /coq/def/Postcond.v:686).

Misc. CheckCFG is implemented in /coq/def/Validator.v:76 and /coq/def/Validator.v:129.
CheckInit is implemented in /coq/def/Validator.v:247, and its specification is proved in /coq/proof/SimulationValid.v:1314.
CheckIncl is implemented in /coq/def/Hints.v:187, and the soundness of (Incl) is proved in /coq/proof/SoundInfrules.v:279.

I Non-Arithmetic Inference Rules

In order to support mem2reg, gvn, and lin2cm, we use 9 non-arithmetic inference rules. In the Coq development, we define the 9 rules in coq/def/Infrules.v:392, and formally verified their soundness in coq/proof/SoundInfrules.v:52.
(transitivity)
\[
\frac{e_{\text{src}} \sqsupset e'_{\text{src}} \quad e'_{\text{src}} \sqsupset e''_{\text{src}}}{e_{\text{src}} \sqsupset e''_{\text{src}}}
\]

(transitivity_tgt)
\[
\frac{e_{\text{tgt}} \sqsupset e'_{\text{tgt}} \quad e'_{\text{tgt}} \sqsupset e''_{\text{tgt}}}{e_{\text{tgt}} \sqsupset e''_{\text{tgt}}}
\]

(substitute)
\[
\frac{vT_{\text{src}} \sqsupset vT'_{\text{src}}}{e_{\text{src}} \sqsupset e_{\text{src}}[vT_{\text{src}} \mapsto vT'_{\text{src}}]}
\]

(substitute_rev)
\[
\frac{vT_{\text{src}} \sqsupset vT'_{\text{src}}}{e_{\text{src}}[vT'_{\text{src}} \mapsto vT_{\text{src}}] \sqsupset e_{\text{src}}}
\]

(substitute_tgt)
\[
\frac{vT_{\text{tgt}} \sqsupset vT'_{\text{tgt}}}{e_{\text{tgt}} \sqsupset e_{\text{tgt}}[vT_{\text{tgt}} \mapsto vT'_{\text{tgt}}]}
\]

(intro_ghost)
\[
\frac{e_{\text{src}} \sim e_{\text{tgt}} \quad e_{\text{tgt}} \sqsupset e_{\text{tgt}}[\hat{g}_{\text{src}}, \hat{g}_{\text{tgt}} \mapsto e_{\text{tgt}}] \quad \hat{g} \text{ not used}}{e_{\text{src}} \sim e_{\text{tgt}}}
\]

(reduce_maydiff_non_physical)
\[
\frac{rT_{\text{src}} \sim rT'_{\text{tgt}}}{e_{\text{src}} \sim e'_{\text{tgt}} \quad e'_{\text{src}} \sqsupset rT_{\text{src}} \quad rT_{\text{src}} \sim rT'_{\text{tgt}}}
\]

\[
\begin{array}{l}
\text{(intro_eq_tgt)} \\
\frac{e_{\text{tgt}} \sqsupset e_{\text{tgt}}}{e_{\text{tgt}} \sqsupset e_{\text{tgt}}}
\end{array}
\]

Each of the rules is based on one of the proof rules in Fig. 10. Here, \(rT_{\text{src}} \sim rT'_{\text{tgt}}\) in premises means \(rT_{\text{src}} \sim P \quad rT'_{\text{tgt}}\) for the current assertion \(P\). Also, \(e_{\text{src}} \sim e'_{\text{tgt}}\) means \(e\) and \(e'\) are of the same expression kind, e.g., they are both add, and for all matching operands \(rT, rT'\), \(rT_{\text{src}} \sim P \quad rT'_{\text{tgt}}\) holds for the current assertion \(P\). \(rT_{\text{src}} \sim rT_{\text{tgt}}\) in conclusions means you can remove \(rT\) from the maydiff set.