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

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

Sungkeun Cho
skcho@ropas.snu.ac.kr
Seoul National University, Korea

Joonwon Choi
joonwonc@mit.edu
MIT, USA

Chung-Kil Hur†
gil.hur@sf.snu.ac.kr

Kwangkeun Yi
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). This result is notable because, to the best of our knowledge, no previous systematic approaches including random testing have found any bugs in the `mem2reg` and `gvn` passes. Moreover, except for the two bugs we have reported, we found only one confirmed miscompilation bug for `mem2reg` in the LLVM bug tracker history.

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

* The first three authors contributed equally to this work and are listed alphabetically.
† Hur is the corresponding author.

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA
2018.

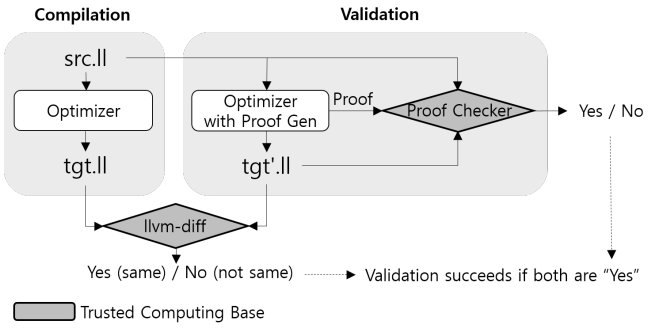


Figure 1. The CRELLVM Framework

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 (`mem2reg` 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 no previous systematic approaches including random testing have found any bugs in the `mem2reg` and `gvn` passes. Moreover, except for the two bugs we have reported, we found only one confirmed miscompilation bug for `mem2reg` 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 `mem2reg` 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.ll` to the target IR code `tgt.ll`. 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'.ll` together with the proof `Proof`. Then the proof checker validates `Proof` to see whether `src.ll` is correctly translated to `tgt'.ll`. 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.ll` and `tgt'.ll` 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.ll` may have unnamed IR registers, `tgt'.ll` 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 `mem2reg` bugs during the verification of inference rules. See the example below.

```
p := alloca(); r := *p
foo(r)                          ⇝ foo(1 / ((int)G - (int)G))
*p := 1 / ((int)G - (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 `mem2reg` pass assumes that constant expressions never raise any undefined behavior such as division-by-zero, which is not true since `1 / ((int)G - (int)G)` forms a valid constant expression in LLVM. Following the logic of `mem2reg`, we also added such a custom inference rule, which we found unsound during the verification of the rule.

[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.

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 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.

```
p := alloca()
loop {                          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.

```
q1 := (p + 10) inbounds       q1 := (p + 10) inbounds
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.

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 := add x 2` is replaced by `y := add a 3` at line 20. This translation can be beneficial because after it, the register `x` may no longer be used and thus `x := add a 1` at line 10 may be eliminated later. This translation is also sound because (*i*) the assertion "`x = add a 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][8]; and (*ii*) from this, we can infer that `add x 2 = add (add a 1) 2 = add a 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.

[4] `undef` 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.

[8] The SSA property says that for every used register x, there is statically (*i.e.,* syntactically) exactly one instruction that defines x (*i.e.,* assigning a value to x), which moreover comes before every use of x.

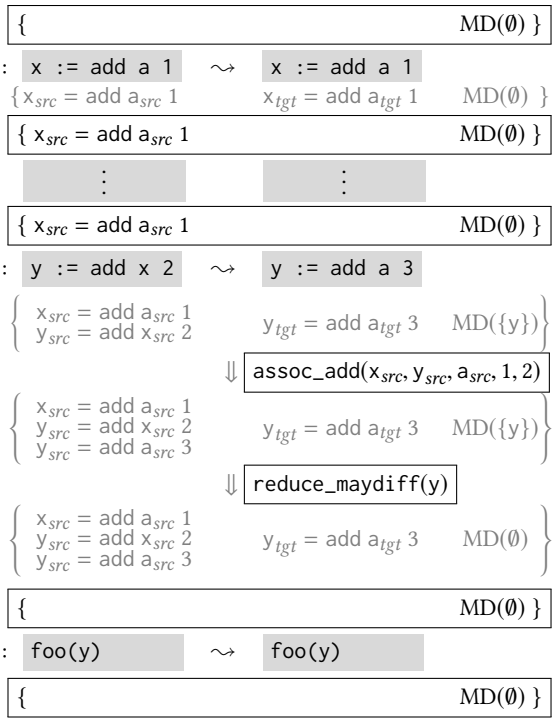


Figure 2. Validation of an `assoc-add` translation in ERHL

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 $\{\,\mathrm{MD}(\emptyset)\,\}$ and the list of inference rules is ($\mathtt{assoc\_add}(\mathsf{x}_{src}, \mathsf{y}_{src}, \mathsf{a}_{src}, 1, 2)$, $\mathtt{reduce\_maydiff(y)}$).

This ERHL proof captures the assertion and the inference step of the intuitive reasoning above. First, the assertion MD(∅) at every line states that every register contains the same value in the source and target program states. Second, the additional assertion $\mathsf{x}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 1$ between line 10 and line 20 states that in the source state, the value of the register x is equal to the result of `add a 1`. Finally, the inference rules $\mathtt{assoc\_add}(\mathsf{x}_{src}, \mathsf{y}_{src}, \mathsf{a}_{src}, 1, 2)$ and 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, $\mathsf{x}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 1$ is a source assertion and $\mathsf{x}_{tgt} = \mathsf{add}\ \mathsf{a}_{tgt}\ 3$ 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 $\mathrm{MD}(M)$:

$$\mathrm{MD}(M) \iff \forall x \notin M.\ x_{src} = x_{tgt}\ .$$

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 $\{\,\mathrm{MD}(\emptyset)\,\}$ 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, $\{\,\mathsf{x}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 1, \mathsf{x}_{tgt} = \mathsf{add}\ \mathsf{a}_{tgt}\ 1, \mathrm{MD}(\emptyset)\,\}$, 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, $\{\,\mathsf{x}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 1, \mathrm{MD}(\emptyset)\,\}$, follows from the computed strong post-assertion by a simple inclusion check.

At line 20, the checker also computes a strong post-assertion, $\{\,\mathsf{x}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 1, \mathsf{y}_{src} = \mathsf{add}\ \mathsf{x}_{src}\ 2, \mathsf{y}_{tgt} = \mathsf{add}\ \mathsf{a}_{tgt}\ 3, \mathrm{MD}(\mathsf{y})\,\}$. 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

Algorithm 1 AssocAdd(F: Function)

A1: **for** l_2: y := add (reg x) (const C_2) **in** F **do**
A2: **if** FindDef(F, x) **is** l_1: x := add (reg a) (const C_1) **then**
A3: C := Simplify(add C_1 C_2)
A4: ReplaceAt(F, l_2, y := add (reg a) (const C))
A5: Assn(x_{src} = add a_{src} C_1, l_1, l_2)
A6: Inf(assoc_add($x_{src}, y_{src}, a_{src}, C_1, C_2$), l_2)
A7: **end if**
A8: **end for**
A9: Auto(reduce_maydiff)

proof checker applies the inference rules given by the proof. The rule $\texttt{assoc\_add}(\mathsf{x}_{src}, \mathsf{y}_{src}, \mathsf{a}_{src}, 1, 2)$ derives $\mathsf{y}_{src} = \text{add } \mathsf{a}_{src}\ 3$ from $\mathsf{x}_{src} = \text{add } \mathsf{a}_{src}\ 1$ and $\mathsf{y}_{src} = \text{add } \mathsf{x}_{src}\ 2$ by associativity:

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

The rule reduce_maydiff(y) removes the register y from the maydiff set because $\mathsf{y}_{src} = \text{add } \mathsf{a}_{src}\ 3$, $\mathsf{y}_{tgt} = \text{add } \mathsf{a}_{tgt}\ 3$ and a is not in the maydiff set:

$$\frac{(\texttt{reduce\_maydiff}(y, e)) \quad y_{src} = e_{src} \quad e_{tgt} = y_{tgt} \quad \text{no registers in } e \text{ are in the maydiff set}}{\text{remove } y \text{ from the maydiff set}}$$

Then, the assertion after line 20, $\{\ \text{MD}(\emptyset)\ \}$, 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 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 l_2: y := add x C_2 with C_2 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 l_1: x := add a C_1 with C_1 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 $C = C_1 + C_2$ and replace the instruction at l_2 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 x_{src} = add a_{src} C_1 at every line between l_1 and l_2. In Fig. 2, the assertion x_{src} = add a_{src} 1 is added at every line between 10 and 20. **[Line A6]:** Add the inference rule $\texttt{assoc\_add}(x_{src}, y_{src}, a_{src}, C_1, C_2)$ at line l_2. In Fig. 2, the rule $\texttt{assoc\_add}(\mathsf{x}_{src}, \mathsf{y}_{src}, \mathsf{a}_{src}, 1, 2)$ 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 $\mathcal{I}$ such that the result of applying $\mathcal{I}$ 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 p is promoted to register accesses to p1 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 p are removed (ignore lnop for now), and every use of the result of a load from p is replaced by the value stored in *p at the time of the load. For example, in Fig. 3, the compiler figures out that *p contains 42 at line 20 (and so does the register a) due to the store of 42 in *p at line 11, and thus replaces the use of a with 42 at line 21. This translation is

[9] The instruction that defines x is unique thanks to the SSA property.

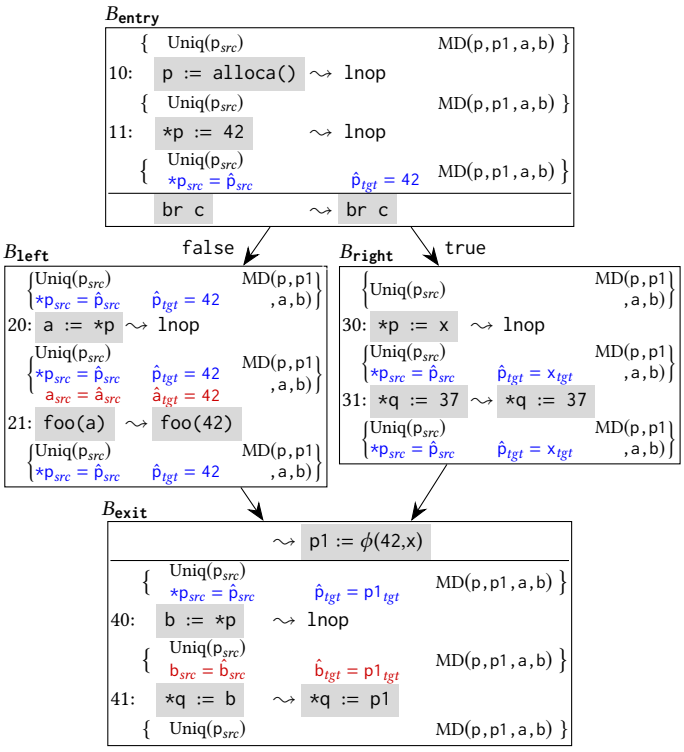


Figure 3. A register promotion example

sound because (*i*) the assertion $*p_{src} = 42$ holds from line 11 to 20; and (*ii*) $a_{src} = 42$ holds from line 20 to 21. Note that we use the blue color for assertions about *p and the red color about the registers containing the value loaded from *p.

In a case where the value stored in *p depends on the control flow, the compiler inserts a phinode, which is a unique construct 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 $\mathbf{B}_{\texttt{left}}$, and x if it comes from $\mathbf{B}_{\texttt{right}}$. In this case, the compiler inserts a phinode p1 := ϕ(42,x) at the beginning of $\mathbf{B}_{\texttt{exit}}$, which defines p1 to be 42 when coming from $\mathbf{B}_{\texttt{left}}$ and x when coming from $\mathbf{B}_{\texttt{right}}$. Then, the use of the register b containing the loaded value from *p can be replaced by p1 at line 41.

3.2 ERHL Proof

We show how to turn the intuition for soundness into a formal ERHL proof, which is given in the unshaded part of Fig. 3 including lnop. Here we omit the inference rules for simplicity, which will be shown later. We introduce interesting features of ERHL by explaining each part of the proof.

Logical No-Ops for Instruction Alignment Logical No-ops, denoted lnop, can be inserted as a part of proof in order to align source and target instructions when their alignment is broken by a translation. For example, in Fig. 3, lnop is inserted at lines 10, 11, 20, 30, 40 because the instructions there are removed by mem2reg.

Note that lnop 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 $\hat{g}$:

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

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 $\{*p_{src} = \hat{p}_{src}, \hat{p}_{tgt} = p1_{tgt}, \mathrm{MD}(p, p1, a, b)\}$ 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 $\hat{(-)}$ 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 $(\hat{rs}_{src}, \hat{rs}_{tgt})$ such that the pair of σ_{src} extended with $\hat{rs}_{src}$ and σ_{tgt} extended with $\hat{rs}_{tgt}$ 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 $\mathbf{B}_{\texttt{left}}$, $a_{src} = 42_{tgt}$ between line 20 and line 21, $*p_{src} = x_{tgt}$ between line 30 and the end of $\mathbf{B}_{\texttt{right}}$, $*p_{src} = p1_{tgt}$ between the beginning of $\mathbf{B}_{\texttt{exit}}$ and line 41, and $b_{src} = p1_{tgt}$ between line 40 and line 41. It is easy to see that these assertions 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 $\mathrm{Uniq}(p_{src})$ at every line. It means that in the source, if p contains an address ℓ, (*i*) ℓ is not aliased with any address stored in the other registers and the memory (*i.e.,* they point to disjoint memory blocks); and (*ii*) ℓ 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(p_{src}), MD({ p, p1, a, b }) }, holds for initial states. It accepts the assertion Uniq(p_{src}) 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 allows an allocation, p := alloca(), 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 p_{src} because p_{src} is updated, (*ii*) adding { Uniq(p_{src}), $*p_{src}$ = undef } because p contains a newly allocated address, and then (*iii*) adding p to the maydiff set. Thus, we have { Uniq(p_{src}), $*p_{src}$ = 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 $*p_{src}$ is private (*i.e.,* has no corresponding target address) due to Uniq(p_{src}) in the pre-assertion. In this case, it computes a post-assertion by (*i*) removing all and only the assertions containing $*p_{src}$ because $*p_{src}$ is updated and p_{src} has no alias with any other address due to Uniq(p_{src}), and then (*ii*) adding { $*p_{src} = x_{src}$ }. Thus, we have { Uniq(p_{src}), $*p_{src} = x_{src}$, MD({ p, p1, a, b }) }.

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

Phinodes At the phinode of $\mathbf{B}_{\texttt{exit}}$, the proof checker validates the assertion separately for each incoming block. For the incoming block $\mathbf{B}_{\texttt{left}}$, the proof checker computes a post-assertion by (*i*) removing all assertions containing $p1_{tgt}$ because $p1_{tgt}$ is updated, (*ii*) adding $42 = p1_{tgt}$ because p1 := 42 is executed in the target when control comes from $\mathbf{B}_{\texttt{left}}$, and then (*iii*) adding p1 to the maydiff set. Then the proof gives the inference rule transitivity($\hat{p}_{tgt}$, 42, $p1_{tgt}$), which derives $\hat{p}_{tgt} = p1_{tgt}$, from which the assertion after the phinode follows trivially. For the incoming block $\mathbf{B}_{\texttt{right}}$, validation succeeds similarly, where the proof gives the inference rule transitivity($\hat{p}_{tgt}$, x_{tgt}, $p1_{tgt}$).

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 b_{src} because b_{src} is updated, (*ii*) adding $b_{src} = *p_{src}$ and then (*iii*) adding b to the maydiff set. Thus, we have {Uniq(p_{src}), $*p_{src} = \hat{p}_{src}$, $\hat{p}_{tgt} = p1_{tgt}$, $b_{src} = *p_{src}$, MD({p, p1, a, b})}.

At this point, the proof gives the rule intro_ghost($\hat{b}$, $\hat{p}$), which adds { $\hat{p}_{src} = \hat{b}_{src}$, $\hat{b}_{tgt} = \hat{p}_{tgt}$ } because $\hat{p}$ is not in the maydiff set. Then the proof gives appropriate transitivity rules, which derives $b_{src} = *p_{src} = \hat{p}_{src} = \hat{b}_{src}$ and $\hat{b}_{tgt} = \hat{p}_{tgt} = p1_{tgt}$, 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 ({ $a_{src} = \hat{a}_{src}$, $\hat{a}_{tgt} = 42$ } at line 21, { $b_{src} = \hat{b}_{src}$, $\hat{b}_{tgt} = p1_{tgt}$ } 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.,* $p \perp q$) in case $*q$ is updated by the store instruction. At lines 21, 31 and 41, thanks to Uniq(p_{src}), the assertions about $*p_{src}$ are preserved.

Note that in the example of Fig. 3, it suffices to use Priv(p_{src}) instead of Uniq(p_{src}). 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(p_{src}) for each promoted location *p*. Also for the sake of performance, we use Uniq instead of introducing $\perp$ 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

Algorithm 2 RegisterPromotion(*F*:Function)

A1: **for** l_a: p:=`alloca()` **in** F **if** p's uses are loads/stores only **do**
A2: InsertEmptyPhinodesFor(F, p)
// Add the phinodes to the maydiff set globally
A3: Remove(l_a), Nop(l_a,*tgt*),Assn({Uniq(p_{src}),MD(p)},*global*)
A4: Inf(`intro_ghost`($\hat{p}$,`undef`),l_a)
A5: WL := [(Entry(F), `undef`, l_a)], MarkVisited(Entry(F))
A6: **while** NonEmpty(WL) **do**
A7: (B, v, l) :: WL := WL
A8: **for** (l_i : i) **in** Instr(B) **do**
A9: **if** i **is** a store instruction ($*p$:= w) **then**
A10: Remove(l_i), Nop(l_i, *tgt*),Inf(`intro_ghost`($\hat{p}$,w),l_i)
A11: v := w, l := l_i
A12: **else if** i **is** a load instruction (x := $*p$) **then**
A13: Assn({$*p_{src} = \hat{p}_{src}, \hat{p}_{tgt} = v_{tgt}$}, l, l_i)
A14: **for** (l_j : j) **in** Use(x) **do**
A15: Replace(F, l_j, x, v), Assn({$x_{src} = \hat{x}_{src}, \hat{x}_{tgt} = v_{tgt}$},$l_i$,$l_j$)
A16: **end for**
A17: Remove(l_i), Nop(l_i, *tgt*),Assn({MD(x)}, *global*)
A18: Inf(`intro_ghost`($\hat{x}$,$\hat{p}$),l_i)
A19: **end if**
A20: **end for**
A21: **for** B' **in** Successor(B) **do**
A22: **if** B' has a phinode ($z := \phi(\cdots)$) inserted at line A2 **then**
A23: z[B] := v, Assn({$*p_{src} = \hat{p}_{src}, \hat{p}_{tgt} = v_{tgt}$}, l, End(B))
A24: **if not** IsVisited(B') **then** WL := (B', z, Begin(B')) :: WL
A25: **else**
A26: **if not** IsVisited(B') **then** WL := (B', v, l) :: WL
A27: **end if**
A28: MarkVisited(B')
A29: **end for**
A30: **end while**
A31: **end for**
A32: Auto(`transitivity`)

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[10], and add them to the maydiff set globally (*i.e.,* at each line). **[Line A3]:** We also remove the allocation, insert `lnop` at that line, and add Uniq(p_{src}) and MD(p) globally. **[Line A4]:** In addition, we add the rule `intro_ghost`($\hat{p}$,`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 { $*p_{src} = \hat{p}_{src}, \hat{p}_{tgt}$ = `undef` } at line l_a, which will be inferred with the help of `intro_ghost`($\hat{p}$,`undef`).

For example, in Fig. 3, the empty phinode p1 := $\phi(-,-)$ is inserted in $\mathbf{B}_{\texttt{exit}}$ and p1 is added to the maydiff set globally; and then the allocation at line 10 is removed, `lnop` is inserted, Uniq(p_{src}) is added and p is added to the maydiff set globally; and finally `intro_ghost`(p̂,`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`, line 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, $\mathbf{B}_{\texttt{entry}}$, $\mathbf{B}_{\texttt{left}}$, $\mathbf{B}_{\texttt{exit}}$, and $\mathbf{B}_{\texttt{right}}$ 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 `lnop`, adds `intro_ghost`($\hat{p}$, 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 `lnop`; `intro_ghost`(p̂, 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 `lnop`, adds x to the maydiff set and adds the rule `intro_ghost`($\hat{x}$,$\hat{p}$) (lines A17-A18).

For example, in Fig. 3, when i is 20: a := $*$p, the load i is replaced by `lnop`; the use of a is replaced by the current value 42 at line 21; $*\mathsf{p}_{src} = 42_{tgt}$ is added from 11 to 20; $\mathsf{a}_{src} = 42_{tgt}$ is added from 20 to 21; a is added to the maydiff set globally; and `intro_ghost`(â,p̂) 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 ($z := \phi(\cdots)$) 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 $(\mathbf{B}_{\texttt{left}}, \mathbf{B}_{\texttt{exit}})$, the phinode `p1 := φ(−,−)` is updated to `p1 := φ(42,−)` and $(\mathbf{B}_{\texttt{exit}}, \texttt{p1}, \text{Begin}(\mathbf{B}_{\texttt{exit}}))$ is added to the worklist WL. Also $*\texttt{p}_{src} = 42_{tgt}$ is added from line 11 to the end of $\mathbf{B}_{\texttt{left}}$.

- 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 $(\mathbf{B}_{\texttt{entry}}, \mathbf{B}_{\texttt{right}})$, the triple $(\mathbf{B}_{\texttt{right}}, 42, \text{line } 11)$ is added to the worklist.

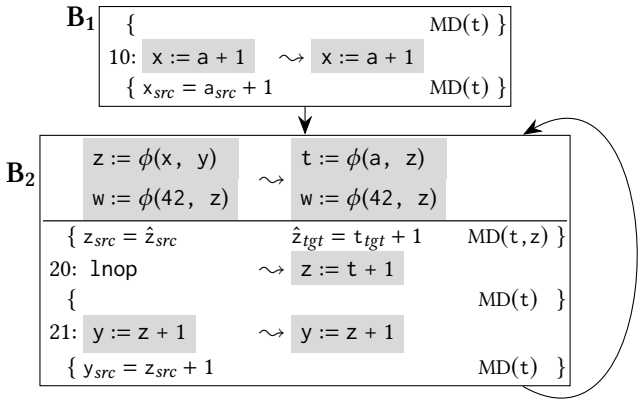
[Line A28]: Finally the successor B' is marked as visited.

Inference Rules As shown in §3.3, the complete proof for `mem2reg` contains two inference rules, `intro_ghost` and `transitivity`. The `intro_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 `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 `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.



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 $\mathbf{B}_2$ 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 $\texttt{z}_{src}$ and $\texttt{z}_{tgt}$ after the phinodes of $\mathbf{B}_2$ and recover it after line 20. The proof makes this point in two steps. It first shows that $\texttt{z}_{src} = \texttt{t}_{tgt} + 1$ holds after the phinodes, and then using it, shows that $\texttt{z}_{src} = \texttt{z}_{tgt}$ holds after line 20. The second step holds trivially and the first step holds as follows. When control comes from $\mathbf{B}_1$, after the phinodes, we have $\texttt{z}_{src} = \bar{\texttt{x}}_{src} = \bar{\texttt{a}}_{src} + 1 = \bar{\texttt{a}}_{tgt} + 1 = \texttt{t}_{tgt} + 1$, where r_{src} denotes the current value of the register r after the phinodes and $\bar{r}_{src}$ the old value of r before the phinodes in the source program; and similarly for r_{tgt} and $\bar{r}_{tgt}$. When control comes from $\mathbf{B}_2$, after the phinodes, we have $\texttt{z}_{src} = \bar{\texttt{y}}_{src} = \bar{\texttt{z}}_{src} + 1 = \bar{\texttt{z}}_{tgt} + 1 = \texttt{t}_{tgt} + 1$.

Proof Validation We show how the ERHL proof checker formally validates the above proof at the most interesting case: the phinodes of $\mathbf{B}_2$ when control comes from itself.

First of all, we need to take into account the old values of registers, as discussed above. For this, we use ghost registers again. Specifically, we reserve a set of ghost registers, denoted $\bar{r}$ and called old registers, for all registers r to represent the old value of r. Note, however, that old registers are just normal ghost registers and technically have nothing to do with physical old values of the corresponding registers.

Now we see how the proof checker uses those old registers at the phinodes of $\mathbf{B}_2$ when control comes from itself. First, it computes a post-assertion from the pre-assertion $\{\, \texttt{y}_{src} = \texttt{z}_{src} + 1, \text{MD}(\texttt{t}) \,\}$ 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
$$\{\, \texttt{y}_{src} = \texttt{z}_{src} + 1, \bar{\texttt{y}}_{src} = \bar{\texttt{z}}_{src} + 1, \text{MD}(\texttt{t}, \bar{\texttt{t}}) \,\}.$$
2. It computes a post-condition from this new assertion as if the assignments `z := ȳ, w := z̄` are executed in the source and `t := z̄, w := z̄` 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 $\bar{\texttt{z}}$ is not in the maydiff set); and (iii) adds the equalities corresponding to the executed assignments. Thus we have
$$\{\, \bar{\texttt{y}}_{src} = \bar{\texttt{z}}_{src} + 1, \texttt{z}_{src} = \bar{\texttt{y}}_{src}, \texttt{w}_{src} = \bar{\texttt{z}}_{src}, \\ \texttt{t}_{tgt} = \bar{\texttt{z}}_{tgt}, \texttt{w}_{tgt} = \bar{\texttt{z}}_{tgt}, \text{MD}(\texttt{t}, \bar{\texttt{t}}, \texttt{z}) \,\}.$$

Then the proof gives the rule $\texttt{intro\_ghost}(\hat{\texttt{z}}, \bar{\texttt{z}} + 1)$, which adds $\{\, \bar{\texttt{z}}_{src} + 1 = \hat{\texttt{z}}_{src}, \hat{\texttt{z}}_{tgt} = \bar{\texttt{z}}_{tgt} + 1 \,\}$ because $\bar{\texttt{z}}$ is not in the maydiff set. Then the automation function adds $\{\, \texttt{z}_{src} = \hat{\texttt{z}}_{src}, \hat{\texttt{z}}_{tgt} = \texttt{t}_{tgt} + 1 \,\}$ by transitivity: $\texttt{z}_{src} = \bar{\texttt{y}}_{src} = \bar{\texttt{z}}_{src} + 1 = \hat{\texttt{z}}_{src}$ and $\hat{\texttt{z}}_{tgt} = \bar{\texttt{z}}_{tgt} + 1 = \texttt{t}_{tgt} + 1$. Finally, all the old registers are completely removed and the assertion after the phinodes $\{\, \texttt{z}_{src} = \hat{\texttt{z}}_{src}, \hat{\texttt{z}}_{tgt} = \texttt{t}_{tgt} + 1, \text{MD}(\texttt{t},\texttt{z}) \,\}$ 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_{src}, Prg_{tgt} and a translation proof Φ, and tries to deduce $Prg_{src} \sim Prg_{tgt}$ using the (SIM) rule. Here, $Entry(F)$ denotes the entry block of the function F; $Prg[F].\zeta[B, i]$

$$\frac{\boxed{Prg_{src} \sim Prg_{tgt}} \quad \text{CheckCFG}(Prg_{src}, Prg_{tgt}) \quad \forall F \in Prg_{src}.\ \text{CheckInit}(\Phi[F].\alpha[\text{Entry}(F), 0]) \quad \forall F,B,i.\ \{\Phi[F].\alpha[B,i]\}\ Prg_{src}[F].\zeta[B,i] \sim Prg_{tgt}[F].\zeta[B,i]\ \{\Phi[F].\alpha[B,i+1]\} \quad \forall F,B,B'.\ \{\Phi[F].\alpha[B,-1]\}\ Prg_{src}[F].\phi[B,B'] \sim Prg_{tgt}[F].\phi[B,B']\ \{\Phi[F].\alpha[B',0]\}}{Prg_{src} \sim Prg_{tgt}}\ (\textsc{Sim})$$

$$\frac{\boxed{\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}} \quad \text{CheckEquivBeh}(P, I_{src}, I_{tgt}) \quad Q = \text{CalcPostAssn}(P, I_{src}, I_{tgt})}{\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}}\ (\textsc{PostAssn}) \qquad \frac{\{P\}\ I_{src} \sim I_{tgt}\ \{Q\} \quad Q \Rrightarrow Q'}{\{P\}\ I_{src} \sim I_{tgt}\ \{Q'\}}\ (\textsc{Consequence})$$

$$\frac{\boxed{Q \Rrightarrow Q'} \quad Q \Rrightarrow Q' \quad Q' \Rrightarrow Q''}{Q \Rrightarrow Q''}\ (\textsc{Trans}) \qquad \frac{rule \in \text{CustomRules} \quad Q' = \text{ApplyInf}(rule, Q)}{Q \Rrightarrow Q'}\ (\textsc{ApplyInf}) \qquad \frac{\text{CheckIncl}(Q, Q')}{Q \Rrightarrow Q'}\ (\textsc{Incl})$$

Figure 4. Proof Rules of ERHL

the i-th instruction of the block B in F; and $Prg[F].\phi[B, B']$ the assignment instructions of the phinodes of B' when control comes from B (*e.g.,* in the source program in §4, $Prg[F].\phi[\mathbf{B_1}, \mathbf{B_2}]$ = { z := x, t := 42 }). Also, $\Phi[F].\alpha[B, i]$ denotes the assertion in the proof Φ just before the i-th instruction of B in F (it denotes the last assertion when $i = -1$).

The checker first checks if Prg_{src} and Prg_{tgt} have the identical CFG (CheckCFG), the assertion in the entry is satisfied by the initial states for each function (CheckInit), and the Hoare triple $\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}$ is valid for all matching intra-block commands I_{src} and I_{tgt} and their pre- and post-assertions P and Q given by Φ. For example, in Fig. 2, it checks at line 20 if $\{\mathtt{x}_{src} = a_{src} + 1, MD(\emptyset)\}$ y := x + 2 ~ y := a + 3 $\{MD(\emptyset)\}$ is valid. It also checks for each inter-block edge from B to B' that $\{P\}\ Prg_{src}.\phi[B, B'] \sim Prg_{tgt}.\phi[B, B']\ \{Q\}$ is valid, where P is the last assertion in B and Q is the first assertion in B'.

To validate a Hoare triple $\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}$, the checker first computes a post-assertion Q_0 with $\{P\}\ I_{src} \sim I_{tgt}\ \{Q_0\}$ using the rule PostAssn (see Appendix H for the definition of CheckEquivBeh and CalcPostAssn). Then it suffices to validate $Q_0 \Rrightarrow Q$ by the rule Consequence.

Then, using the rules ApplyInf and Trans, the checker iteratively applies a sequence of inference rules $rule_1, \ldots, rule_n$ (either given by Φ or generated by automation function) and deduces $Q_0 \Rrightarrow Q_n$, where $Q_i = \text{ApplyInf}(rule_i, Q_{i-1})$.

Finally, the checker validates $Q_n \Rrightarrow Q$ using the rule Incl, where CheckIncl performs a simple inclusion check.

Semantic Interpretation For the soundness of the proof checker, we give the semantic interpretation of the top-level judgment as semantics preservation, or behavior refinement:

$$[\![Prg_{src} \sim Prg_{tgt}]\!] \stackrel{\text{def}}{=} Beh(Prg_{src}) \supseteq Beh(Prg_{tgt})\ .$$

The soundness of (Sim) is proved using a local simulation in the style of [14], which is a simplification of parametric bisimulation [13]. First, we show that CheckInit(P) implies:

$$\forall \sigma_{src}, \sigma_{tgt}, \alpha.\ FInit(\sigma_{src}) \wedge FInit(\sigma_{tgt}) \implies [\![P]\!]_\alpha(\sigma_{src}, \sigma_{tgt})\ .$$

Here, $FInit(\sigma)$ means σ is a possible initial state of a function call, $[\![P]\!]$ is the semantic interpretation of the assertion P (see Appendix G for details), and α 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:

$$\begin{aligned}[\![\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}]\!] \stackrel{\text{def}}{=} \forall \sigma_{src}.\ Instr(\sigma_{src}) = I_{src} \implies \\ \forall \sigma_{tgt}.\ Instr(\sigma_{tgt}) = I_{tgt} \implies \\ \forall \alpha, \sigma'_{tgt}, \varepsilon.\ [\![P]\!]_\alpha(\sigma_{src}, \sigma_{tgt}) \wedge \sigma_{tgt} \xrightarrow{\varepsilon} \sigma'_{tgt} \implies \\ \exists \sigma'_{src}, \alpha'.\ [\![Q]\!]_{\alpha'}(\sigma'_{src}, \sigma'_{tgt}) \wedge \sigma_{src} \xrightarrow{\varepsilon} \sigma'_{src} \wedge \alpha \sqsubseteq \alpha'\end{aligned}$$

where, $Instr(\sigma)$ is the next instruction to execute in the program state σ, and $\sigma \xrightarrow{\varepsilon} \sigma'$ means the state σ steps to σ' emitting an observable event ε. Also, $\sqsubseteq$ is the extension relation of memory injection.

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

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

$$\begin{aligned}[\![Q \Rrightarrow Q']\!] \stackrel{\text{def}}{=} \forall \sigma_{src}, \sigma_{tgt}, \alpha.\ [\![Q]\!]_\alpha(\sigma_{src}, \sigma_{tgt}) \implies \\ \exists \alpha'.\ [\![Q']\!]_{\alpha'}(\sigma_{src}, \sigma_{tgt}) \wedge \alpha \sqsubseteq \alpha'\ .\end{aligned}$$

For the soundness of (ApplyInf), every custom *rule* should satisfy that $[\![Q \Rrightarrow \text{ApplyInf}(rule, Q)]\!]$ 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`,

	mem2reg	gvn	licm	instcombine
Compiler (Covered)	568	1,027	706	702
Proof Generation	213	413	286	1,357

Figure 5. SLOC of Proof-Generation Code

and `licm`, including `transitivity` and `intro_ghost`, are formally verified in Coq (see Appendix I for details).

Verification of Proof Checker In order to reduce TCB, we formally verified the soundness of the proof checker in Coq (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.

	Results			Time (sec.)			
	#V	**#F**	**#NS**	**Orig**	**PCal**	**I/O**	**PCheck**
mem2reg	76.62K	10	10.58K	9.04	313.46	8.83K	15.05K
gvn	357.17K	295	7.79K	41.37	245.39	39.09K	33.15K
licm	168.20K	0	24.93K	22.71	853.67	56.68K	11.22K
instcombine	1593.44K	0	528.75K	182.89	432.53	141.53K	90.01K

Figure 6. Experimental Results

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 analysis modules of LLVM; 2,344 (0.41%) alter type declarations; 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 Crellvm 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 9.3 MB in total (5.4 MB for `mem2reg`, 0.4 MB for `gvn`, 2.2 MB for `licm`, 1.3 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 phinodes 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

[1] [n. d.]. The Coq Proof Assistant. https://coq.inria.fr/. ([n. d.]). Accessed: 2017-11-16.

[2] [n. d.]. HOL Interactive Theorem Prover. https://hol-theorem-prover.org/. ([n. d.]). Accessed: 2017-11-16.

[3] [n. d.]. LLVM Linux. http://llvm.linuxfoundation.org. ([n. d.]). Accessed: 2017-11-16.

[4] [n. d.]. The SPEC CINT2006 Benchmark. https://www.spec.org/cpu2006/CINT2006/. ([n. d.]). Accessed: 2017-11-16.

[5] [n. d.]. The Z3 Theorem Prover. https://github.com/Z3Prover/z3. ([n. d.]). Accessed: 2017-11-16.

[6] Andrew W. Appel. 2001. Foundational Proof-Carrying Code (LICS '01).

[7] Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. ACM Trans. Program. Lang. Syst. 36, 1 (March 2014).

[8] Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations (POPL '04).

[9] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers (PLDI '13).

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Trans. Program. Lang. Syst. 13, 4 (Oct. 1991).

[11] Delphine Demange, David Pichardie, and Léo Stefanesco. 2016. Verifying Fast and Sparse SSA-Based Optimizations in Coq (CC '16).

[12] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. 2013. Will You Still Compile Me Tomorrow? Static Cross-version Compiler Validation (ESEC/FSE '13).

[13] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The Marriage of Bisimulations and Kripke Logical Relations. In POPL.

[14] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A Formal C Memory Model Supporting Integer-pointer Casts (PLDI '15).

[15] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML (POPL '14).

[16] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs (PLDI '14).

[17] Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant (POPL '06).

[18] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* (2009).

[19] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. The CompCert Memory Model, Version 2. Research report RR-7987. INRIA.

[20] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive (PLDI '15).

[21] David Menendez and Santosh Nagarakatte. 2017. Alive-Infer: Data-driven Precondition Inference for Peephole Optimizations in LLVM (PLDI '17).

[22] David Menendez, Santosh Nagarakatte, and Aarti Gupta. 2016. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM (SAS '16).

[23] Kedar S. Namjoshi, Giacomo Tagliabue, and Lenore D. Zuck. 2013. A Witnessing Compiler: A Proof of Concept (RV '13).

[24] Kedar S. Namjoshi and Lenore D. Zuck. 2013. Witnessing Program Transformations (SAS '13).

[25] George C. Necula. 1997. Proof-carrying Code (POPL '97).

[26] George C. Necula. 2000. Translation Validation for an Optimizing Compiler (PLDI '00).

[27] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. 2014. Global Sparse Analysis Framework. ACM Trans. Program. Lang. Syst. 36, 3 (Sept. 2014).

[28] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation (TACAS '98).

[29] Amir Pnueli, Ofer Strichman, and Michael Siegel. 1998. The Code Validation Tool CVT: Automatic Verification of a Compilation Process (STTT '98).

[30] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs (PLDI '12).

[31] Silvain Rideau and Xavier Leroy. 2010. Validating Register Allocation and Spilling (CC '10).

[32] Martin C. Rinard and Darko Marinov. 1999. Credible Compilation with Pointers (RRV '99).

[33] Hanan Samet. 1978. Proving the Correctness of Heuristically Optimized Code (ACM '78).

[34] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-based Translation Validator for LLVM (CAV '11).

[35] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization (POPL '09).

[36] Zachary Tatlock and Sorin Lerner. 2010. Bringing Extensibility to Verified Compilers (PLDI '10).

[37] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-graph Translation Validation for LLVM (PLDI '11).

[38] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations (POPL '08).

[39] Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified Validation of Lazy Code Motion (PLDI '09).

[40] Jean-Baptiste Tristan and Xavier Leroy. 2010. A Simple, Verified Validator for Software Pipelining (POPL '10).

[41] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers (PLDI '11).

[42] Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product (FM '08).

[43] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations (POPL '12).

[44] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-based Optimizations for LLVM (PLDI '13).

[45] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. 2002. Translation and Run-Time Validation of Loop Transformations (RV '02).

[46] Lenore D. Zuck, Amir Pnueli, and Benjamin Goldberg. 2003. VOC: A Methodology for the Translation Validation of OptimizingCompilers (J. UCS '03).

	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	1	6.95K	0	0	2.50K	0	448	15.97K	0	1.50K
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	348	0	258	337	0	269	1.04K	0	782
464.h264ref	51.58K	532	1	0	12.71K	27	0	8.77K	0	1.49K	22.45K	0	5.31K
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	1.05K	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	381	89.11K	0	13.09K
gimp-2.8.18	1004.20K	19.45K	6	528	37.01K	22	313	24.37K	0	3.85K	150.89K	0	44.14K
ghostscript-9.14.0	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	7.99K	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

	Time (sec.)															
	mem2reg				gvn				licm				instcombine			
	Orig	PCal	I/O	PCheck	Orig	PCal	I/O	PCheck	Orig	PCal	I/O	PCheck	Orig	PCal	I/O	PCheck
400.perlbench	0.27	14.20	0.19K	0.79K	1.37	7.95	1.44K	4.05K	0.27	10.72	0.90K	0.33K	5.67	12.67	6.02K	9.06K
401.bzip2	0.01	1.46	0.01K	0.13K	0.08	0.47	0.13K	0.21K	0.05	1.44	0.08K	0.03K	0.36	1.19	0.24K	0.32K
403.gcc	0.75	39.44	0.58K	2.90K	5.49	31.92	3.71K	4.99K	1.53	28.21	3.08K	0.57K	18.67	57.40	19.07K	19.81K
429.mcf	<0.01	0.10	<0.01K	<0.01K	0.01	0.06	<0.01K	<0.01K	0.01	0.03	<0.01K	<0.01K	0.07	0.07	<0.01K	<0.01K
433.milc	0.03	0.91	0.01K	0.02K	0.10	0.48	0.08K	0.04K	0.08	0.35	0.07K	0.02K	0.44	0.55	0.11K	0.05K
445.gobmk	0.19	7.55	1.92K	0.92K	0.67	3.71	0.45K	0.44K	0.31	3.40	0.35K	0.09K	2.95	3.85	1.84K	0.94K
456.hmmer	0.07	2.75	0.03K	0.08K	0.31	1.54	0.11K	0.09K	0.18	1.54	0.16K	0.05K	1.49	1.87	0.23K	0.14K
458.sjeng	0.02	0.74	0.01K	0.02K	0.15	0.69	0.08K	0.06K	0.06	0.50	0.04K	0.01K	0.58	0.81	0.15K	0.14K
462.libquantum	0.01	0.27	<0.01K	<0.01K	0.03	0.13	<0.01K	<0.01K	0.01	0.10	<0.01K	<0.01K	0.14	0.22	0.01K	<0.01K
464.h264ref	0.11	6.78	0.06K	0.39K	0.71	4.58	1.16K	0.95K	0.35	8.37	1.08K	0.47K	2.68	5.98	1.64K	1.04K
470.lbm	<0.01	0.20	<0.01K	<0.01K	0.01	0.04	<0.01K	<0.01K	0.02	0.05	<0.01K	<0.01K	0.03	0.04	<0.01K	<0.01K
482.sphinx3	0.04	1.40	0.02K	0.03K	0.16	0.74	0.05K	0.04K	0.09	0.57	0.06K	0.02K	0.83	1.01	0.15K	0.09K
sendmail-8.15.2	0.15	2.82	0.02K	<0.01K	0.55	2.49	0.49K	0.38K	0.22	3.71	0.46K	0.17K	2.24	3.52	1.47K	0.77K
emacs-25.1	0.66	34.80	1.12K	2.30K	2.83	22.44	4.71K	3.18K	0.93	23.91	4.12K	0.95K	15.21	35.06	20.14K	8.94K
python-3.4.1	0.89	33.92	1.18K	0.86K	3.84	24.69	4.24K	2.04K	0.79	17.16	3.55K	0.77K	16.04	26.01	13.06K	3.78K
gimp-2.8.18	1.62	47.49	1.06K	0.80K	5.11	28.06	2.03K	1.02K	2.05	19.51	2.25K	0.46K	29.52	39.52	5.68K	2.39K
ghostscript-9.14.0	2.15	25.37	0.32K	0.06K	7.81	40.08	4.50K	3.08K	4.58	47.70	5.46K	1.57K	28.01	51.34	13.05K	6.07K
LLVM nightly test	2.06	93.27	2.30K	5.76K	12.15	75.32	15.90K	12.60K	11.21	686.41	35.01K	5.72K	57.96	191.44	58.67K	36.47K
Total	9.04	313.46	8.83K	15.05K	41.37	245.39	39.09K	33.15K	22.71	853.67	56.68K	11.22K	182.89	432.53	141.53K	90.01K

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);
}
```

```
void foo(int arr[], int n) {
  int sqrsum = 0, diffsqrsum = 0;
  int i, prev, cur;
  for (i = 0; i < n; ++i) {
    prev = cur; cur = arr[i];
    sqrsum += sqr(i, prev, cur);
    diffsqrsum += diffsqr(i, prev, cur);
  }
  printf ("square sum=%d, diff sqr sum=%d \n", sqrsum, diffsqrsum);
}
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.,* :

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

The function `foo()` calculates the two summations in a single loop. Note that the function `diffsqr(i, prev, cur)` returns $(\texttt{cur} - \texttt{prev})^2$ if $\texttt{i} > 0$, 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 `diffsqr()` 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.

$$VT = [\texttt{x1}, \texttt{x2} \mapsto ①; \quad \texttt{y1}, \texttt{y2}, \texttt{y3} \mapsto ②]$$
$$ET = [n - 2 \mapsto ①; \quad 1 + ① \mapsto ②]$$
$$LT = [\ldots; \quad \mathbf{B}_{\texttt{empty}} \mapsto [① \mapsto \texttt{x1};\ ② \mapsto \texttt{10}]; \quad \mathbf{B}_{\texttt{right}} \mapsto [① \mapsto \texttt{x2};\ ② \mapsto \texttt{y2}]; \quad \ldots]$$

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 ① to `x1` and `x2`, and ② to `y1`, `y2` and `y3`; and the expression table ET assigns value number ① to the expression $n - 2$, and ② to $1 + ①$. This means that, at any point of execution, there exist some values for ① and ② such that `x1`, `x2` and $n - 2$ evaluate to the value ① and `y1`, `y2`, `y3` and $1 + ①$ evaluate to the value ② whenever

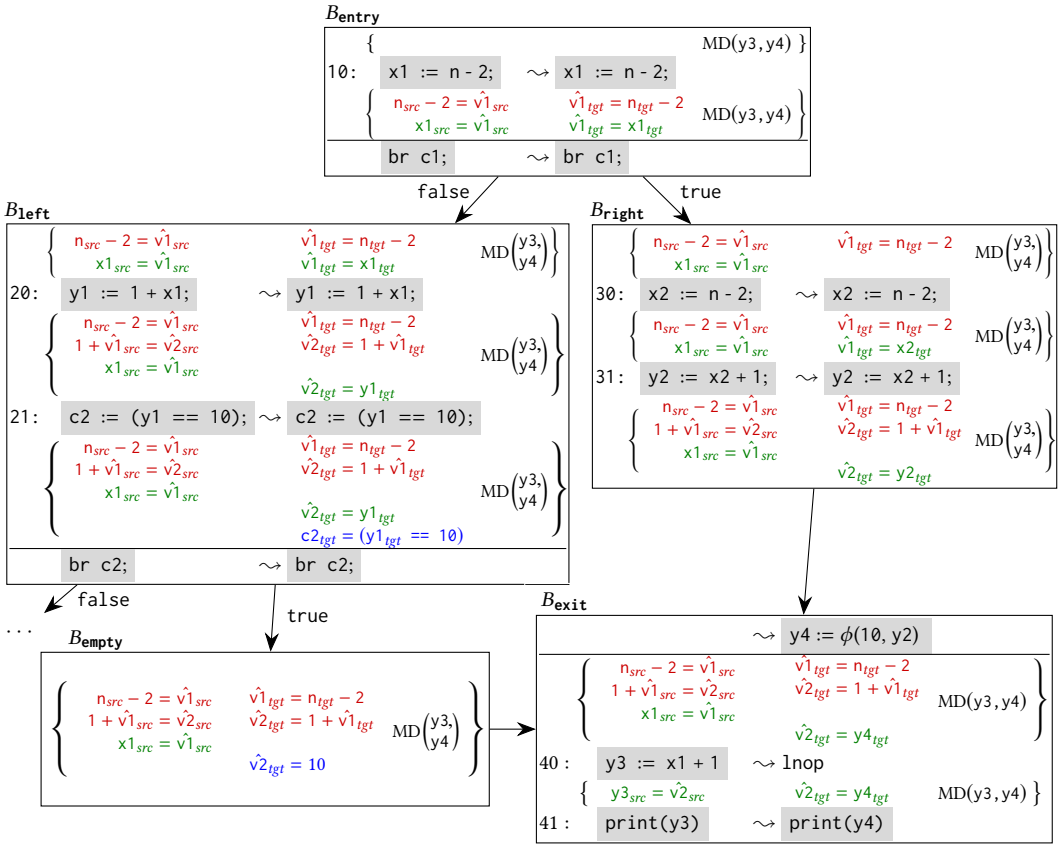


Figure 9. A PRE Example

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 $\mathbf{B}_{\texttt{entry}}$, the leader value for ② does not exist because none of the values with value number ② are defined in the block.

In the example, PRE detects partial redundancies of the instruction `y3 := x1 + 1` in $\mathbf{B}_{\texttt{exit}}$ because `y3` belongs to ② and $\mathbf{B}_{\texttt{exit}}$'s incoming blocks have a leader value for ②: `10` in $\mathbf{B}_{\texttt{empty}}$ and `y2` in $\mathbf{B}_{\texttt{right}}$. In other words, depending on the control flow, the expression `y3` is equivalent to either `10` or `y2`. To eliminate the redundant instruction, PRE (i) inserts the phinode `y4 := φ(10,y2)` at the beginning of $\mathbf{B}_{\texttt{exit}}$; ($ii$) replaces all uses of `y3` with `y4` at line 41; and (iii) eliminates `y3 := x1 + 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.

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 $e_{src} = \hat{v}i_{src}$ or $\hat{v}i_{tgt} = e_{tgt}$ 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 ① and ②) 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 $x_{src} = \hat{v}i_{src}$ or $\hat{v}i_{tgt} = x_{tgt}$ relate values with their value numbers according to the value table VT. For example, the assertion after line 21 also states that `x1` has the value $\hat{v}1$ in the source and `y1` has the value $\hat{v}2$ in the target.

Branching Assertions The blue equalities show properties derived from branching conditions. For example, the equality $\texttt{c2}_{tgt} = (\texttt{y1}_{tgt} == 10)$ at line 21 is derived from the definition of the branching register `c2`, and the equality $\hat{v}2_{tgt} = 10$ in the block $\mathbf{B}_{\texttt{empty}}$ is derived from the fact that `c2` must be true in $\mathbf{B}_{\texttt{empty}}$ and thus `y1` 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 $\{\ \mathtt{y3}_{src} = \mathtt{x1}_{src} + 1\ \}$, and deduces, by applying the inference rules from the proof, that:

$$\begin{aligned}
\mathtt{y3}_{src} &= \mathtt{x1}_{src} + 1 && \text{(post-assertion)}\\
&= \hat{\mathtt{v1}}_{src} + 1 && (\mathtt{substitution}(\mathtt{x1}_{src} + 1, \mathtt{x1}_{src}, \hat{\mathtt{v1}}_{src}))\\
&= 1 + \hat{\mathtt{v1}}_{src} && (\mathtt{commutativity_add}(\hat{\mathtt{v1}}_{src}, 1))\\
&= \hat{\mathtt{v2}}_{src} && \text{(expression assertion)}
\end{aligned}$$

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 $\{\ \mathtt{y2}_{src} = \mathtt{x2}_{src} + 1, \mathtt{x2}_{tgt} + 1 = \mathtt{y2}_{tgt}\ \}$. Then the proof gives the inference rule $\mathtt{intro\_ghost}(1 + \hat{\mathtt{v1}}, \hat{\mathtt{v2}})$, which adds $\{\ 1 + \hat{\mathtt{v1}}_{src} = \hat{\mathtt{v2}}_{src}, \hat{\mathtt{v2}}_{tgt} = 1 + \hat{\mathtt{v1}}_{tgt}\ \}$. Then the proof checker deduces $\hat{\mathtt{v2}}_{tgt} = \mathtt{y2}_{tgt}$ similarly as in line 40.

Adding Phinode Assertions At the phinode of $\mathbf{B}_{\mathtt{exit}}$, the assertion is separately validated for each incoming block. For the incoming block $\mathbf{B}_{\mathtt{left}}$ (and similarly for $\mathbf{B}_{\mathtt{right}}$), the proof checker computes a post-assertion by adding $\mathtt{10} = \mathtt{y4}_{tgt}$ and adding y to the maydiff set, and derives $\hat{\mathtt{v2}}_{tgt} = \mathtt{10} = \mathtt{y4}_{tgt}$ by applying the transitivity rule from the proof.

Adding Branching Assertions At line 21, the proof checker computes a post-assertion by adding $\{\ \mathtt{c2}_{src} = (\mathtt{y1}_{tgt}\ \mathtt{==}\ \mathtt{10}), \mathtt{c2}_{tgt} = (\mathtt{y1}_{tgt}\ \mathtt{==}\ \mathtt{10})\ \}$, from which the next assertion trivially follows.

Using Branching Assertions At the beginning of $\mathbf{B}_{\mathtt{empty}}$, the proof checker computes a post-assertion by adding $\{\ \mathtt{true} = \mathtt{c2}_{src}, \mathtt{true} = \mathtt{c2}_{tgt}\ \}$ from the branching condition, and deduces, by applying the inference rules from the proof, that $\mathtt{true} = \mathtt{c2}_{tgt} = (\mathtt{y1}_{tgt}\ \mathtt{==}\ \mathtt{10})$ and

$$\begin{aligned}
\hat{\mathtt{v2}}_{tgt} &= \mathtt{y1}_{tgt} && \text{(value assertion)}\\
&= \mathtt{10} && (\text{from } \mathtt{true} = (\mathtt{y1}_{tgt}\ \mathtt{==}\ \mathtt{10}) \text{ by } \mathtt{icmp_to_eq}(\mathtt{true}, \mathtt{y1}_{tgt}, \mathtt{10}))
\end{aligned}$$

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:

$$\begin{aligned}
RET &= [\mathtt{x1}, \mathtt{x2} \mapsto \{\ n_{src} - 2 = \hat{\mathtt{v1}}_{src}, \hat{\mathtt{v1}}_{tgt} = n_{tgt} - 2\ \};\\
&\qquad \mathtt{y1}, \mathtt{y2}, \mathtt{y3}, \mathtt{y4} \mapsto \{\ n_{src} - 2 = \hat{\mathtt{v1}}_{src}, \hat{\mathtt{v1}}_{tgt} = n_{tgt} - 2, 1 + \hat{\mathtt{v1}}_{src} = \hat{\mathtt{v2}}_{src}, \hat{\mathtt{v2}}_{tgt} = 1 + \hat{\mathtt{v1}}_{tgt}\ \}]\\
RPT &= [\mathtt{x1}, \mathtt{x2} \mapsto (_, _);\quad \mathtt{y1} \mapsto (_, ①);\quad \mathtt{y2}, \mathtt{y3} \mapsto (①, _);\quad \mathtt{y4} \mapsto (②, ②)]\\
BCT &= [(\mathtt{10}, ②) \mapsto \{\ (\mathbf{B}_{\mathtt{left}}, \mathtt{true}, \mathtt{y1})\ \}]\ .
\end{aligned}$$

The register expression table *RET* contains sufficient expression assertions for reasoning about each register. For example, $RET(\mathtt{x1})$ contains $\{\ n_{src} - 2 = \hat{\mathtt{v1}}_{src}, \hat{\mathtt{v1}}_{tgt} = n_{tgt} - 2\ \}$, 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, $RPT[\mathtt{y1}]$ contains (_, ①), which means that the second parameter x1 of the instruction 20: y1 := 1 + x1 has the value number ①. The branching condition table *BCT* maps pairs of constants/function parameters and their value numbers to their associated branching information. For example, $BCT[(\mathtt{10}, ②)]$ contains $(\mathbf{B}_{\mathtt{left}}, \mathtt{true}, \mathtt{y1})$, which means that (*i*) y1 = 10 holds when control flows from $\mathbf{B}_{\mathtt{left}}$ 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** $(l', y, \hat{m})$ **in** MatchExpr(e, $RPT[x]$) **do**
A9: $WL := (l', y, \hat{m}, side) :: WL$
A10: **end for**
A11: | Some (ConstantOrParameter(C)) $\Rightarrow$
A12: **match** FindBranchingCondition($BCT[(C, \hat{n})]$, l_0, F) **with**
A13: | Some $(v, y, c, l) \Rightarrow$
A14: **if** FindDef(F, c) = Some $(l_c: c := e)$ **then** Assn($c_{side} = e_{side}$, l_c, l) **end if**
A15: Inf(`icmp_to_eq`(v,y_{tgt},C), l)
A16: Assn($RET[y]$, l, l_0), Assn($C_{side} = \hat{n}_{side}$, l, l_0)
A17: $WL := (l, y, \hat{n}, side) :: WL$
A18: **end match**
A19: **end match**
A20: **end while**
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 = `y3` is defined at line 40 in the source, so the initial works are (40, `y3`, $\hat{v2}$, src) and (40, `y4`, $\hat{v2}$, 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`, $\hat{v2}$, tgt), for example. The register `y4` is defined by the phinode of $\mathbf{B}_{\texttt{exit}}$, so l is the phinode of $\mathbf{B}_{\texttt{exit}}$ and $e = \phi(10, \texttt{y2})$. Hence $RET[\texttt{y4}]$ and $\hat{v2}_{tgt} = \texttt{y4}_{tgt}$ are inserted from the phinode of $\mathbf{B}_{\texttt{exit}}$ to line 40. For the work (40, `y3`, $\hat{v2}$, 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 $\texttt{y3}_{src} = \hat{\texttt{v2}}_{src}$ after line 40, $\texttt{x1}_{src} = \hat{\texttt{v1}}_{src}$ should hold before the line, so the code adds (40, `x1`, $\hat{\texttt{v1}}$, src) to the worklist; for justifying $\hat{\texttt{v2}}_{tgt} = \texttt{y4}_{src}$ after the phinode of $\mathbf{B}_{\texttt{exit}}$, the code adds ([edge from $\mathbf{B}_{\texttt{empty}}$ to $\mathbf{B}_{\texttt{exit}}$]), 10, $\hat{v2}$, tgt) and ([edge from $\mathbf{B}_{\texttt{right}}$ to $\mathbf{B}_{\texttt{exit}}$], `y2`, $\hat{v2}$, 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, $\hat{v2}$, tgt), where E_e is the edge from $\mathbf{B}_{\texttt{empty}}$ to $\mathbf{B}_{\texttt{exit}}$. The code finds a branching condition in $BCT[(10, \hat{v2})]$ which guarantees that $10 = \hat{v2}$ holds at E_e (line A12). Block $\mathbf{B}_{\texttt{left}}$ 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 $\mathbf{B}_{\texttt{left}}$ to $\mathbf{B}_{\texttt{empty}}$). Hence the code adds $\texttt{c2}_{tgt} = (\texttt{y1}_{tgt} == \texttt{10})$ from line 21 to E_l (line A14), and adds the inference rule `icmp_to_eq`(`true`,$\texttt{y1}_{tgt}$,`10`) at E_l (line A15). Then the code adds the expression assertions $RET[\texttt{y1}]$ for `y1` and the branching assertion $\hat{\texttt{v2}}_{tgt} = \texttt{10}$ from E_l to E_e (line A16), and adds the work (E_l, `y1`, $\hat{v2}$, 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`.

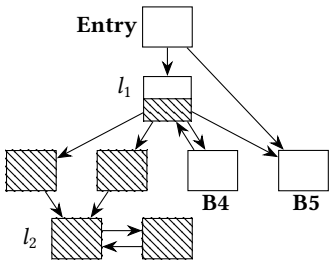
D Micro-Optimizations in `instcombine`

We validated the following 139 micro-optimizations in `instcombine`:
`add-comm-sub`, `add-const-not`, `add-dist-sub`, `add-mask`, `add-onebit`, `add-or-and`, `add-select-zero`, `add-shift`, `add-signbit`, `add-sub`, `add-xor-and`, `add-zext-bool`, `and-de-morgan`, `and-mone`, `and-not`, `and-or-const2`, `and-or-not1`, `and-or`, `and-same`, `and-undef`, `and-xor-const`, `and-zero`, `bitcast-bitcast`, `bitcast-fpext`, `bitcast-fptosi`, `bitcast-fptoui`, `bitcast-fptrunc`, `bitcast-inttoptr`, `bitcast-ptrtoint`, `bitcast-sametype`, `bitcast-sext`, `bitcast-sitofp`, `bitcast-trunc`, `bitcast-uitofp`, `bitcast-zext`, `bop-associativity`, `dead-code-elim`, `dead-store-elim`, `fold-phi-bin-const`, `fold-phi-bin`, `fpext-bitcast`, `fpext-fpext`, `fptosi-bitcast`, `fptosi-fpext`, `fptoui-bitcast`, `fptoui-fpext`, `fptrunc-bitcast`, `fptrunc-fpext`, `icmp-eq-add-add`, `icmp-eq-srem`, `icmp-eq-sub-sub`, `icmp-eq-sub`, `icmp-eq-xor-not`, `icmp-eq-xor-xor`, `icmp-ne-add-add`, `icmp-ne-srem`, `icmp-ne-sub-sub`, `icmp-ne-sub`, `icmp-ne-xor-xor`, `icmp-ne-xor`, `icmp-sge-or-not`, `icmp-sgt-and-not`, `icmp-sle-or-not`, `icmp-slt-and-not`, `icmp-swap`, `icmp-uge-or-not`, `icmp-ugt-and-not`, `icmp-ule-or-not`, `icmp-ult-and-not`, `inttoptr-bitcast`, `inttoptr-ptrtoint`, `mul-bool`, `mul-mone`, `mul-neg`, `mul-shl`, `or-and-xor`, `or-and`, `or-mone`, `or-not`, `or-or2`, `or-or`, `or-same`, `or-undef`, `or-xor2`, `or-xor3`, `or-xor4`, `or-xor`, `or-zero`, `ptrtoint-bitcast`, `ptrtoint-inttoptr`, `sdiv-mone`, `select-bop-fold`, `select-icmp-eq-xor1`, `select-icmp-eq-xor2`, `select-icmp-eq`, `select-icmp-gt-const`, `select-icmp-lt-const`, `select-icmp-ne-xor1`, `select-icmp-ne-xor2`, `select-icmp-ne`, `select-icmp-sgt-xor1`, `select-icmp-sgt-xor2`, `select-icmp-slt-xor1`, `select-icmp-slt-xor2`, `sext-bitcast`, `sext-sext`, `sext-trunc-ashr`, `sext-zext`, `shift-undef1`, `shift-undef2`, `shift-zero1`, `shift-zero2`, `sitofp-bitcast`, `sitofp-sext`, `sitofp-zext`, `sub-add`, `sub-const-add`, `sub-const-not`, `sub-mone`, `sub-onebit`, `sub-or-xor`, `sub-remove`, `sub-shl`, `sub-sub`, `trunc-bitcast`, `trunc-onebit`, `trunc-sext`, `trunc-trunc`, `trunc-zext`, `uitofp-bitcast`, `uitofp-zext`, `xor-same`, `xor-undef`, `xor-zero`, `zext-bitcast`, `zext-trunc-and-xor`, `zext-trunc-and`, `zext-xor`, `zext-zext`

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

E Program Points between Two Lines

Assertion(P, l_1, l_2) in the proof generation code means predicate P should be added to the assertions between l_1 and l_2. More specifically, the proposition P should be added at every program point appearing in a path from l_1 to l_2 that does not visit l_1 but may visit l_2 in-between. Since l_1 is the source of the proposition P so that we can get P as a post-assertion every time we visit l_1, there is no need to add P along a path from l_1 to l_1. For example, consider the following program.



The marked area between l_1 and l_2 is where we should add the proposition P.

Thanks to the SSA property [10], we can efficiently calculate the program points between l_1 and l_2. First, we can assume that l_1 dominates l_2 (*i.e.,* one should have visited l_1 to reach l_2 from the entry point), since the proposition P created at l_1 should hold at l_2. Then a program point l is on a path from l_1 to l_2 that does not visit l_1 in-between if and only if (i) l_1 dominates l and (ii) l_2 is reachable from l without visiting l_1. We efficiently check the first condition using the dominator tree [10] and the second condition by a backward BFS search from l_2.

In the above example, any program point in the marked area is such that l_1 dominates it and l_2 is reachable from it without visiting l_1. For example, l_2 is not reachable from the block B_4 without visiting l_1, and l_1 does not dominate the block B_5.

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 has 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 $\mathsf{y}_{src} = \mathsf{a}_{src} - \mathsf{x}_{src} = \mathsf{a}_{src} - (\mathsf{a}_{src} - 1) = 1$ holds for any integer value of

a_{src}:

$$\begin{array}{l} \mathsf{x := a - 1;} \\ \mathsf{y := a - x;} \\ \mathsf{z := y + 1;} \end{array} \quad \rightsquigarrow \quad \begin{array}{l} \mathsf{x := a - 1;} \\ \mathsf{y := a - x;} \\ \mathsf{z := 1 + 1;} \end{array}$$

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

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

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

G Semantic Interpretation of Assertions

The syntax of assertions is as follows:

$$\begin{aligned} \mathit{Reg} \ni r &::= \cdots \\ \mathit{Const} \ni c &::= \cdots \\ \mathit{Typ} \ni \mathit{typ} &::= \mathbf{int} \mid *\,\mathit{typ} \mid \cdots \\ \mathit{SVal} \ni v &::= r \mid c \\ \mathit{Tag} \ni \mathit{tag} &::= \mathit{Phy} \mid \mathit{Ghost} \mid \mathit{Old} \\ \mathit{Reg} \times \mathit{Tag} \ni r\mathit{T} &::= (r, \mathit{tag}) \\ \mathit{SVal} \times \mathit{Tag} \ni v\mathit{T} &::= (v, \mathit{tag}) \\ \mathit{Expr} \ni e &::= \mathsf{add}\ v\mathit{T}\ v\mathit{T} \mid \mathsf{load}\ v\mathit{T}\ \mathit{typ}\ \mathit{align} \mid \cdots \\ \mathit{Pred} \ni \mathit{pred} &::= e \sqsupseteq e \mid \mathrm{Uniq}(r) \mid \mathrm{Priv}(r\mathit{T}) \mid v\mathit{T} \perp v\mathit{T} \\ \mathit{MD} \ni M &::= \{r\mathit{T}, \cdots\} \\ \mathit{AssnU} \ni S, T &::= \{\mathit{pred}, \cdots\} \\ \mathit{Assn} \ni P, Q &::= (S, T, M) \end{aligned}$$

Reg, *Const*, and *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 *Phy* means physical registers, *Ghost* means regular ghost registers (§3.2), and *Old* means old ghost registers (§4). We write r for (r, Phy), $\hat{r}$ for (r, Ghost), and $\bar{r}$ for (r, Old).

An expression is basically 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[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 $e_1 \sqsupseteq e_2$ means either $e_1 = e_2$ or $e_1 = \mathsf{undef}$, and the predicate $v\mathit{T}_1 \perp v\mathit{T}_2$ means the addresses in $v\mathit{T}_1$ and $v\mathit{T}_2$ 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:

$$\begin{aligned} \mathit{Val} \ni V &::= \cdots \\ \mathit{RF} \ni \mathit{rs} &::= \{r \mapsto V, \cdots\} \\ \mathit{State} \ni \sigma &::= \cdots \\ \mathit{StateT} \ni \sigma\mathit{T} &::= (\sigma, \mathit{rs}, \mathit{rs}) \\ \mathit{Meminj} \ni \alpha &::= \cdots \end{aligned}$$

[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:

$$\begin{aligned}
[\![r]\!], [\![c]\!], [\![v]\!] &: State \to Val \stackrel{\text{def}}{=} \cdots \\
[\![rT]\!], [\![vT]\!] &: StateT \to Val \stackrel{\text{def}}{=} \cdots \\
\\
[\![pred]\!] &: 2^{Blk} \to StateT \to \mathbb{P} \\
[\![e_1 \sqsupseteq e_2]\!](priv, \sigma T) &\stackrel{\text{def}}{=} [\![e_1]\!](\sigma T) \sqsupseteq [\![e_2]\!](\sigma T) \\
[\![\text{Uniq}(r)]\!](priv, \sigma T) &\stackrel{\text{def}}{=} \forall b, b', o, o'.\ [\![r]\!](\sigma T) = (b, o) \land (b', o') \in \sigma(T \setminus r) \implies b \neq b' \\
[\![\text{Priv}(rT)]\!](priv, \sigma T) &\stackrel{\text{def}}{=} \forall b, o.\ [\![rT]\!](\sigma T) = (b, o) \implies b \in priv \\
[\![vT_1 \perp vT_2]\!](priv, \sigma T) &\stackrel{\text{def}}{=} \forall b_1, b_2, o_1, o_2.\ [\![vT_1]\!](\sigma T) = (b_1, o_1) \land [\![vT_2]\!](\sigma T) = (b_2, o_2) \implies b_1 \neq b_2 \\
\\
[\![M]\!](\alpha, \sigma T_{src}, \sigma T_{tgt}) &\stackrel{\text{def}}{=} \forall rT \notin M.\ [\![rT]\!](\sigma T_{src}) \sim_\alpha [\![rT]\!](\sigma T_{tgt}) \\
[\![S]\!](priv, \sigma T) &\stackrel{\text{def}}{=} \forall pred \in S.\ [\![pred]\!](priv, \sigma T) \\
[\![(S, T, M)]\!](\alpha, \sigma_{src}, \sigma_{tgt}) &\stackrel{\text{def}}{=} \exists \sigma T_{src}, \sigma T_{tgt}. [\![M]\!](\alpha, \sigma T_{src}, \sigma T_{tgt}) \land \\
&\qquad \sigma T_{src}.0 = \sigma_{src} \land [\![\sigma]\!](priv_{src}(\alpha), \sigma T_{src}) \land \\
&\qquad \sigma T_{tgt}.0 = \sigma_{tgt} \land [\![T]\!](priv_{tgt}(\alpha), \sigma T_{tgt})
\end{aligned}$$

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 alised with any other values in σT (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 points 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 $I_{src} = (x_{src} := \texttt{call}\ f_{src}\ args_{src})$ and $I_{tgt} = (x_{tgt} := \texttt{call}\ f_{tgt}\ args_{tgt})$ as follows:

$$\begin{aligned}
[\![\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}]\!] \stackrel{\text{def}}{=} & \\
\forall \sigma_{src}, \sigma_{tgt}, \alpha.\ & [\![P]\!]_\alpha(\sigma_{src}, \sigma_{tgt}) \land Instr(\sigma_{src}) = I_{src} \land Instr(\sigma_{tgt}) = I_{tgt} \implies \\
& (f_{src} \sim_\alpha f_{tgt}) \land (args_{src} \sim_\alpha args_{tgt}) \land \\
\forall v_{src}, v_{tgt}, \sigma'_{src}, \sigma'_{tgt}, \alpha' \sqsupseteq \alpha.\ & [\![\top]\!]_{\alpha'}(\sigma'_{src}, \sigma'_{tgt}) \land v_{src} \sim_{\alpha'} v_{tgt} \land \sigma_{src} \stackrel{v_{src}}{\Rightarrow} \sigma'_{src} \land \sigma_{tgt} \stackrel{v_{tgt}}{\Rightarrow} \sigma'_{tgt} \implies \\
& \exists \alpha'' \sqsupseteq \alpha'.\ [\![Q]\!]_{\alpha''}(\sigma'_{src}, \sigma'_{tgt})\ .
\end{aligned}$$

where, $v_{src} \sim_\alpha v_{tgt}$ means v_{src} is injected into v_{tgt} via memory injection α, $\top$ is the assertion with no predicates and the full maydiff set (*i.e.,* the values of every register may differ), and $\sigma \stackrel{v}{\Rightarrow} \sigma'$ means σ is about to call a function, and σ' is a possible return state after the function call for the case that the callee returns v.

Algorithm 4 CheckEquivBeh(P, I_{src}, I_{tgt}: Command)

match I_{src}, I_{tgt} **with**
// call
| (x_{src} := `call` f_{src} $args_{src}$), (x_{tgt} := `call` f_{tgt} $args_{tgt}$) ⇒ **return** ($f_{src} \sim_P f_{tgt}$) ∧ ($args_{src} \sim_P args_{tgt}$)
| (_ := `call` _ _), _ | _, (_ := `call` _ _) ⇒ **return** `false`
// alloca
| (p_{src} := `alloca` x_{src}), (p_{tgt} := `alloca` x_{tgt}) ⇒ **return** $x_{src} \sim_P x_{tgt}$
| (p_{src} := `alloca` x_{src}), (`lnop`) ⇒ **return** `true`
| (_ := `alloca` _), _ | _, (_ := `alloca` _) ⇒ **return** `false`
// store
| (`store` p_{src} v_{src}), (`store` p_{tgt} v_{tgt}) ⇒ **return** ($p_{src} \sim_P p_{tgt}$) ∧ ($v_{src} \sim_P v_{tgt}$)
| (`store` p_{src} v_{src}), (`lnop`) ⇒ **return** Priv(p_{src}) ∈ P
| (`store` _), _ | _, (`store` _) ⇒ **return** `false`
// target is load
| (v_{src} := `load` p_{src}), (v_{tgt} := `load` p_{tgt}) ⇒ **return** $p_{src} \sim_P p_{tgt}$
| _, (v_{tgt} := `load` p_{tgt}) ⇒ **return** `false`
// target is div
| (_ := `div` _ b_{src}), (_ := `div` _ b_{tgt}) ⇒ **return** $b_{src} \sim_P b_{tgt}$
| _, (_ := `div` _ b_{tgt}) ⇒ **return** IsNonzero(P, b_{tgt})
// misc.
| _, _ ⇒ **return** `true`
end match

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

P' := Prune(P, I_{src}, I_{tgt})
(P''_{src}, P''_{tgt}, M'') := AddMemoryPreds(I_{src}, I_{tgt}, P')
P'''_{src} := AddLessdefPreds(I_{src}, P''_{src})
P'''_{tgt} := AddLessdefPreds(I_{tgt}, P''_{tgt})
return ReduceMaydiff(P'''_{src}, P'''_{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 α' of the current memory injection α, if the return values and return states are related by α', then there exists a future extension α'' of α' 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, $x_{src} \sim_P y_{tgt}$ means one of the followings holds: (i) $x = y$ and x is not in the maydiff set; (ii) $(x_{src} \sqsupseteq y_{src}) \in P_{src}$ and y is not in the maydiff set; or (iii) x is not in the maydiff set and $(x_{tgt} \sqsupseteq y_{tgt}) \in P_{tgt}$. Basically, this implies $x_{src} \sim_\alpha y_{tgt}$ holds for all α that is compatible with P. Also, $e_{src} \sim_P e'_{tgt}$ means e and e' are of the same expression kind, *e.g.,* they are both `add`, and for all matching operands x, x', $x_{src} \sim_P x'_{tgt}$ 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 `lnop` 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} := PruneU(P_{src}, I_{src})
3: P'_{tgt} := 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} := `alloca`$(\cdots)$), (p_{tgt} := `alloca`$(\cdots)$), 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} := `alloca`$(\cdots)$) and I_{tgt} is `lnop`, 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} := `call`$(\cdots)$), (x_{tgt} := `call`$(\cdots)$) 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 := op *args*), add ($x \sqsupseteq$ op *args*) and (op *args* $\sqsupseteq x$) (see /coq/def/Postcond.v:936). Note that load is regarded as side-effect-free.
- If I is a store instruction $*p := v$, add $*p \sqsupseteq v$ (see /coq/def/Postcond.v:939).
- If I is an allocation instruction $*p$:= `alloca`$(\cdots)$, add $*p \sqsupseteq$ `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/Postcond.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 $x \sqsupseteq \bar{y}, \bar{y} \sqsupseteq x$ 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/SoundImplies.v:279.

I Non-Arithmetic Inference Rules

In order to support `mem2reg`, `gvn`, and `licm`, 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)
$$\frac{e_{src} \sqsupseteq e'_{src} \qquad e'_{src} \sqsupseteq e''_{src}}{e_{src} \sqsupseteq e''_{src}}$$

(transitivity_tgt)
$$\frac{e_{tgt} \sqsupseteq e'_{tgt} \qquad e'_{tgt} \sqsupseteq e''_{tgt}}{e_{tgt} \sqsupseteq e''_{tgt}}$$

(substitute)
$$\frac{v_{src} \sqsupseteq v'_{src}}{e_{src} \sqsupseteq e_{src}[v_{src} \mapsto v'_{src}]}$$

(substitute_rev)
$$\frac{v_{src} \sqsupseteq v'_{src}}{e_{src}[v'_{src} \mapsto v_{src}] \sqsupseteq e_{src}}$$

(substitute_tgt)
$$\frac{v_{tgt} \sqsupseteq v'_{tgt}}{e_{tgt} \sqsupseteq e_{tgt}[v_{tgt} \mapsto v'_{tgt}]}$$

(intro_ghost)
$$\frac{e_{src} \sim e_{tgt}}{e_{src} \sqsupseteq \hat{g}_{src}, \hat{g}_{tgt} \sqsupseteq e_{tgt}}\ \hat{g} \text{ not used}$$

(intro_eq_tgt)
$$\frac{}{e_{tgt} \sqsupseteq e_{tgt}}$$

(reduce_maydiff_non_physical)
$$\frac{}{rT_{src} \sim rT_{tgt}}\ \text{rT is not physical (i.e. ghost or old) and not used}$$

(reduce_maydiff_lessdef)
$$\frac{rT_{src} \sqsupseteq e_{src} \quad e_{src} \sim e'_{tgt} \quad e'_{src} \sqsupseteq rT_{src}}{rT_{src} \sim rT_{tgt}}$$

Figure 10. Formally Verified Inference Rules

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