ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 NRF ERC

ROSAEC MEMO
2014-002
September 30, 2014

An Extensible Verified Validator For Simple Optimizations in LLVM

Sungkeun Cho*, Joonwon Choi*, Jeehoon Kang*,
Chung-Kil Hur, and Kwangkeun Yi
{skcho,jwchoi}@ropas.snu.ac.kr, {jeehoon.kang,gil.hur}@sf.snu.ac.kr, kwang@ropas.snu.ac.kr
Seoul National University

September 30, 2014

Abstract

Recent advances in theorem proving technology have made it possible to write a completely bug-free compiler such as CompCert. However, this technology have not been fully applied to mainstream compilers such as GCC and LLVM due to the huge amount of work required. Nevertheless, we believe that the verified validation technique will reduce verification efforts significantly and thus make it feasible to provide a formal guarantee of correctness for the full LLVM compiler.

As a first step towards the grand goal, we have developed an extensible verified validator for simple optimizations in LLVM, mainly targeted for all micro-optimizations (about 400) in the instruction combine pass. Our validator, based on a simple form of relational Hoare reasoning, can be used to validate various kinds of simple optimizations without losing completeness (*i.e.,* it succeeds to validate all correct results at least by design). We achieve both generality and completeness, not by automation, but by extensibility, which allows to freely add inference rules that are proven to be correct.

1 Introduction

Since compilers are indispensable tools for software development, their reliability is crucial, especially for safety-critical software. However, as recent researches [11, 3] have shown, mainstream C compilers such as GCC and LLVM contain plenty of bugs that matter practically. For example, [11] found 79 bugs in GCC and 202 bugs in LLVM, and [3] found 79 bugs in GCC and 68 bugs in LLVM.

In recent years, there has been a considerable amount of work to increase the reliability of a compiler. The main techniques include random testing [11, 3], translation validation [6, 7], verified compilation [4, 8] and verified validation [8, 9, 10]. These approaches have their own advantages and disadvantages. The former two are general-purpose techniques, so that they have been used to find bugs or validate compilation results in mainstream compilers. However, these methods do not provide formal guarantee of absence of bugs. On the other hand, the latter two techniques provide the highest level of guarantee, a formal proof of semantics preservation. However, they have not been fully applied to mainstream C compilers such as GCC and LLVM, due to the huge amount of work required.

The grand goal of our project, called LLVMBerry, is to provide a formal guarantee of semantics preservation for the LLVM compiler, by minimizing the amount of verification effort required. To this end, we will use the verified validation technique only without using any verified compilation, and try to reduce verification efforts by maximizing the merits of the validation approach. As the first step towards our grand goal and as a case study, we have

*The first three authors equally contribuited to this paper and are listed alphabetically.

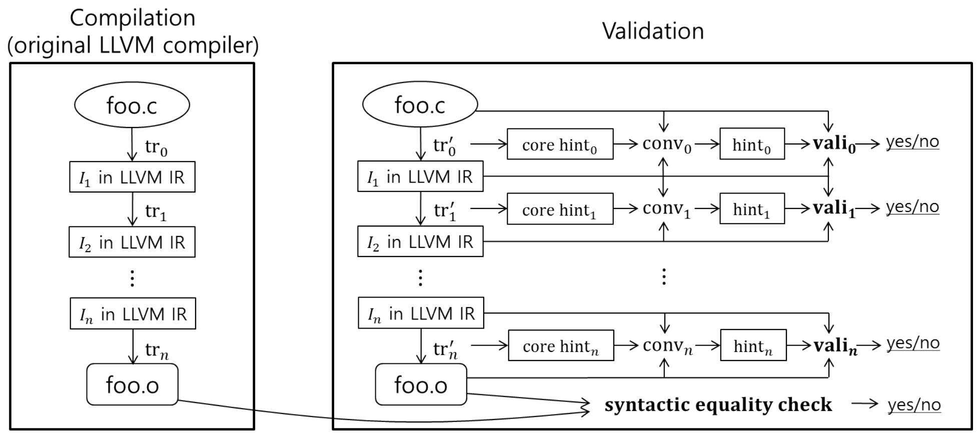


Figure 1: The framework of LLVMBERRY: Programs in boldface need to be verified.

developed a verified validator, which we call SIMPLEBERRY, for numerous micro optimizations in the instruction combine pass of the LLVM compiler, which is the main result of the paper. We elaborate more on these points in the following.

1.1 Verified Validation vs. Verified Compilation

Before we discuss why we chose verified validation over verified compilation, we briefly review the two methods.

Compiler verification consists of two steps: *(i)* to implement a compiler as a computable function $C : \mathrm{Src} \to \texttt{option}\ \mathrm{Tgt}$ in a proof assistant such as Coq, and *(ii)* to prove that every target program produced by the compiler implements only the behaviors specified by the source program:

$$\forall s, t.\ C(s) = \texttt{Some}\ t \implies \mathrm{behaviors}(t) \subseteq \mathrm{behaviors}(s)\ .$$

Then, one can extract the verified compiler function as an executable program and use it to compile source programs.

Verified validation for a particular compiler, introduced by Tristan and Leroy [8, 9, 10], also consists of two steps: *(i)* to implement a translation validator specialized for the compiler as a computable function $V : \mathrm{Src} \times \mathrm{Tgt} \to \texttt{bool}$ in a proof assistant, and *(ii)* to prove that any translation validated by the validator preserves the semantics of the source program:

$$\forall s, t.\ V(s, t) = \texttt{true} \implies \mathrm{behaviors}(t) \subseteq \mathrm{behaviors}(s)\ .$$

Then, one can extract the verified validator as an executable program and use it to check whether a compilation result of the compiler is valid or not.

Verified validation for the LLVM compiler has the following advantages over verified compilation.

1. We do not need to re-implement the compiler in a proof assistant. This is a big advantage because fully implementing the LLVM compiler in a proof assistant like Coq is practically impossible due to the huge amount work required. Even if it is possible, since the verified compiler is extracted into a functional language like OCaml, the verified compiler is hard to be as fast as the original compiler written in C++. This may be a problem because the compilation speed matters in practice.

2. A compiler development team and a verification team can be separated. More specifically, in the verified validation approach, compiler experts can write a compiler in C++ and

verification experts can write a verified validator in a proof assistant, whereas, in the verified compilation approach, those with expertise in both compilation and verification have to write a verified compiler in a proof assistant. This separation is important in practice because not many people have expertise in both area.

3. Verifying a validator requires less verification efforts than verifying a compiler. This is because, in many cases, a validator that checks translations of a particular optimization pass is much simpler than the optimization pass itself. Furthermore, in some cases, a single validator can be reused to validate results of other similar optimization passes.

4. Verified validation is less sensitive to changes to the compiler. Since verified validation does not directly verify the code of a compiler, it may not be much affected by changes to the compiler, such as version upgrades, which regularly occur in mainstream compilers. More specifically, even if an optimization algorithm changes in order to improve the performance, the validation algorithm may remain the same or change slightly.

Now we discuss the downsides of verified validation and how they can be overcome.

1. A verified validator may fail to validate correct translations. According to the aforementioned correctness statement, verification of a validator only guarantees that the validator never validates any incorrect translation. Thus, there is no guarantee that the validator validates all correct translations.

   However, since we develop a validator that is targeted for a particular optimization pass, it is always possible to design the validator to be *complete w.r.t. the target optimizer*, meaning that it validates all correct results that are intended by the optimizer. Although the design may have a bug that leads to incompleteness, such a bug is easily detected by the verified validator. More specifically, if the validator fails to validate a translation result of the target optimizer, there are only two possibilities: *(i)* the translation is incorrect (*i.e.,* the optimizer has a bug); or *(ii)* the validator is incomplete (*i.e.,* the design has a bug). In either case, the details of the validation will guide us to the point where the bug occurs, so that we can easily fix the bug.

2. Validation may take long. Since validation includes checking various invariants and conditions, it may be much slower than compilation. If validation is invoked for every compilation, this will increase the whole compilation time, which can be problematic.

   However, compilation and validation can be completely separated. Thus, a developer can first compile a source program and use the compiled program before the validation of the compilation is completed. Moreover, one can selectively validate compilation results during software development. For example, she can validate a compilation result only when the compiled program behaves erroneously, or when it is a release version.

   Also, validation time can be reduced greatly by using multi-cores or distributed computing since validation is highly parallelizable. A compiler consists of many optimization passes, each of which may be performed many times during compilation. Thus, a single compilation consists of at least hundreds of optimization steps. Since each optimization step can be validated completely independently, validating the whole optimization steps is perfectly suited to multi core and distributed computing environment.

1.2 Our Approach to Verified Validation for LLVM

The framework of LLVMBERRY is depicted in Figure 1. In this framework, we provide a completely separated validation module (RHS of the figure), so that a user can use the original LLVM compiler for compilation (LHS of the figure) and *a posteriori* validate the compilation result whenever she wants.

In the validation module, tr'_i is the original optimization pass tr_i of the LLVM compiler, extended with additional code that generates *core hints* for validation. Core hints are the key information about why the optimizer performed such optimization, and they are used by the associated validator vali_i to determine validity of the optimization result. It is such hints that make it possible to design a *complete* validator for any particular optimizer.

conv_i, written in OCaml, is a *hint converter* that converts core hints in JSON format to data that the validator vali_i can understand. Furthermore, conv_i can also expand core hints to full hints by adding more hints inferable from the core hints together with the input and the output programs. Note that conv_i is not a part of the validator vali_i because conv_i does not need to be verified.

vali_i is a *validator* that takes input and output programs with hints and determines validity of the translation using the hints. Unlike other parts, vali_i has to be verified in order for our validation to be sound. Thus, vali_i is first written and verified in the proof assistant Coq, and then extracted as an executable program in OCaml. The formal soundness statement for vali_i is as follows:

$$\forall s, t, h.\ \mathrm{vali}_i(s, t, h) = \texttt{true} \implies \mathrm{behaviors}(t) \subseteq \mathrm{behaviors}(s).$$

The final part of the validation is to check whether the original output program ('foo.o' in LHS of the figure) and the validated output program ('foo.o' in RHS of the figure) are syntactically equal. Rigorously speaking, this syntactic equality checker also has to be verified, but we just use the GNU "diff" because we trust it for such a simple task.

Our basic principle for minimizing verification efforts is not to care about validation time. In other words, whenever we can reduce verification efforts by increasing validation time, we do it. We think this principle is reasonable because, as we discussed before, validation can be performed off-line and is highly parallelizable.

The following are examples of our specific efforts that would make verification easier, though they may increase validation time.

1. We try to split a translation into as small translations as possible. This would make verification simpler because a validator just needs to focus on a small translation at a time.

2. We try to use the same validator for many different optimizations by designing it as general as possible.

3. We do not prove the preservation of properties such as well-formedness and read-onlyness of a function that can be additional verification overhead for each validator. Instead, we just dynamically check the properties at validation time whenever they are needed. Though we have to verify the correctness of the well-formedness and read-onlyness checker, they are verified once and used many times.

1.3 Extensible Verified Validator for Simple Optimizations

To support our arguments for verified validation of the LLVM compiler, we developed an extensible verified validator, called Simpleberry, and used it to validate micro optimizations in the instruction combine pass of the LLVM version 3.0. For formalization of the LLVM IR (Intermediate Representation), we used the Vellvm [12, 13] framework.

Here are summaries of our work.

- Simpleberry is based on a simple form of relational Hoare reasoning and designed to be general enough to validate many simple optimizations that do not change the control flow graph (CFG) of programs, including all micro optimizations (about 400) in the instruction combine pass.

- We did not perform any complex automation to achieve such generality because such automation would require complex verification. Instead, we designed Simpleberry to be extensible, so that one can freely add user-defined inference rules specific to a particular optimization by just proving the correctness of the rules.
- Examples of optimization that we validated do not only include peephole optimizations, but also simple inter-block, heap and phi-node optimizations.
- As of now, we have validated 50 optimizations out of about 400 optimizations in the instruction combine pass. However, after we built the infrastructure for Simpleberry and tested it for 3 representative optimizations, it took 5 person-weeks to add 47 more optimizations. Adding one micro optimization includes understanding the optimization code, adding and verifying new inference rules, adding code that generates core hints, and sometimes debugging. We expect to add the remaining 350 optimizations within 4 months by two PhD students.

The contributions of the paper are as follows:

1. We designed a framework for verified validation of LLVM compiler.
2. We applied the verified validation approach to a mainstream compiler for the first time.
3. We developed an extensible verified validator for the first time.

2 Overview

In this section, we present the high-level design of Simpleberry.

Since Simpleberry does not essentially rely on the details of LLVM IR such as the SSA property, we will present our work using a simple abstract language, where the syntax and the semantics are intuitively clear. First, we assume that our language has variables (also called registers) $x, y, z, \ldots$ and the heap. We use C-like notation for heap access such as $*x = y$ and $x = *y$. We will explain other details of the language on the fly when we need them.

Simpleberry is basically a proof checker for a simple form of relational Hoare logic. For validation, we first give an invariant at each line of code and ask Simpleberry to check whether these invariants are valid (or consistent). Since checking validity of a Hoare logic is undecidable in the presence of heap, we allow a user to add any sound inference rules in order to increase validation power of Simpleberry.

Simple Relational Invariants First of all, Simpleberry has a fixed heap invariant. Throughout the reasoning, informally speaking, we implicitly keep the following invariant for the heaps h_{src} and h_{tgt} in the source and the target programs:

$$\begin{aligned}&\exists h^{\text{e}}_{\text{src}}, h^{\text{i}}_{\text{src}}, h^{\text{e}}_{\text{tgt}}, h^{\text{i}}_{\text{tgt}}.\\ &\quad (h^{\text{e}}_{\text{src}} \uplus h^{\text{i}}_{\text{src}} \subseteq h_{\text{src}}) \wedge (h^{\text{e}}_{\text{tgt}} \uplus h^{\text{i}}_{\text{tgt}} \subseteq h_{\text{tgt}}) \wedge (h^{\text{e}}_{\text{src}} \approx h^{\text{e}}_{\text{tgt}})\ . \qquad (1)\end{aligned}$$

This says that the source and the target heaps are split into two parts such that one parts are equivalent and the other parts are unrelated. We call the former *public heaps* and the latter *isolated heaps.* (See Section 5 for more formal details.)

Simpleberry allows a user to specify an invariant about variables at each line of code (*i.e.,* pre and post conditions of Hoare logic). An invariant at line i have three components: source property P_i, target property Q_i and may-diff set D_i.

$$\begin{array}{rcccc} & \{\,P_{i-1}\,\} & & \{\,Q_{i-1}\,\} & \{\,D_{i-1}\,\} \\ i: & \text{src command}_i & \mapsto & \text{tgt command}_i & \\ & \{\,P_i\,\} & & \{\,Q_i\,\} & \{\,D_i\,\} \end{array}$$

The source property P_i is a property about the state of the source program that should hold just after the execution of the instruction at line i. Similarly, Q_i is a property that holds at line i in the target program. D_i is what connects the states of the source and the target programs at line i.

Source and target properties P_i and Q_i are sets consisting of five kinds of basic properties: (i) equality about a register value $x = e$ (*e.g.*, $x = y + 1$); (ii) equality about a heap value $*p = v$; (iii) no alias between pointer values $p \neq q$; (iv) allocated pointers $\texttt{alc}(p)$; and (v) isolated pointers $\texttt{isol}(p)$. The semantics of (i) and (ii) is as expected. $p \neq q$ denotes that the values contained in the variables (or registers) p and q are pointer values that point to disjoint parts of the heap. $\texttt{alc}(p)$ denotes that the variable p contains a pointer value that points to an allocated part of the heap. $\texttt{isol}(p)$ denotes that x contains a pointer value that points to the isolated heap.

A may-diff set D_i at line i contains all variables x such that the value of x in the source program and the value of x in the target program may be different at line i. In other words, every variable that is not in D_i should contain equivalent values in the source and the target programs.

Regarding the notion of equivalent values and heaps, we can simply understand them as the same values and heaps for now. Additionally, if equivalent values are pointer values, they should point to the public heaps in the source and the target. Similarly, the values in the public heap $h^{\text{e}}_{\text{src}}$ (resp., $h^{\text{e}}_{\text{tgt}}$) should satisfy that if they are pointer values, they point back to the public heap $h^{\text{e}}_{\text{src}}$ (resp., $h^{\text{e}}_{\text{tgt}}$). More formal details will be presented in Section 5.

Finally, we allow logical (or existential) variables in invariants in order to increase the expressive power. A logical variable $\widehat{x}$ has nothing to do with the physical value of the variable x and thus is distinguished from the variable x. It only denotes the existence of some value and is used to connect several equations. We will explain more details in Section 3.3.

Validation Algorithm Given invariants, Simpleberry checks the consistency of the invariants. At each line, it first conservatively checks whether the source and the target programs raise the same side effects, such as heap update, function call and error. Then, it calculates a post condition $P'_{i-1}, Q'_{i-1}, D'_{i-1}$ that is strong enough for our purpose[1], and tries to check whether $P'_{i-1}, Q'_{i-1}, D'_{i-1}$ implies P_i, Q_i, D_i.

$$\begin{array}{rcccc}
 & \{\, P_{i-1} \,\} & & \{\, Q_{i-1} \,\} & \{\, D_{i-1} \,\} \\
i: & \text{src command}_i & \mapsto & \text{tgt command}_i & \\
 & [\, P'_{i-1} \,] & & [\, Q'_{i-1} \,] & [\, D'_{i-1} \,] \\
 & [\, P''_{i-1} \,] & & [\, Q''_{i-1} \,] & [\, D''_{i-1} \,] \\
 & \{\, P_i \,\} & & \{\, Q_i \,\} & \{\, D_i \,\}
\end{array}$$

Checking the implication is the most difficult task. Here, Simpleberry does not perform any automation but, instead, simply applies the inference rules given as hints to derive a new invariant $P''_{i-1}, Q''_{i-1}, D''_{i-1}$. Finally, it checks whether $P''_{i-1} \supseteq P_i$, $Q''_{i-1} \supseteq Q_i$ and $D''_{i-1} \subseteq D_i$ hold. Note the direction of containment is reversed for may-diff sets because they are properties about those variables not in the may-diff sets.

Extensibility In order to validate a particular optimization, one may need special inference rules for that optimization. Since it is infeasible for Simpleberry to initially have all such inference rules, we allow to register user-defined inference rules. In this way, we can easily extend the power of Simpleberry to validate results of a new optimizer.

We do not have any restriction on the expressibility of user-defined rules. An inference rule can be any computable function, say f, in Coq that takes an invariant (P, Q, D) and returns

[1]It is not strongest because we do not perform any complex automation.

Transformation

B1		
9: $x = a + 1$	$\mapsto$	$x = a + 1$

B4		
5: $y = x + 2$	$\mapsto$	$y = a + 3$

Invariant Commands

9@B1: `propagate_to 5@B4 in src`

Inference Rule Commands

5@B4: `add_assoc` x y `in src`

5@B4: `remove_maydiff` y

Validation

B1			
	$\{\}$	$\{\}$	$\{\}$
9: $\mathbf{x = a + 1}$ $\mapsto$		$\mathbf{x = a + 1}$	
	$[\,x = a + 1\,]$	$[\,x = a + 1\,]$	$[\ ]$
	$\{\,x = a + 1\,\}$	$\{\}$	$\{\}$

B4			
	$\{\,x = a + 1\,\}$	$\{\}$	$\{\}$
5: $\mathbf{y = x + 2}$ $\mapsto$		$\mathbf{y = a + 3}$	
	$[\,x = a + 1, y = x + 2\,]$	$[\,y = a + 3\,]$	$[\,y\,]$
	`add_assoc` x y a `1 2 3 in src`		
	$[\,x = a + 1, y = x + 2, y = a + 3\,]$	$[\,y = a + 3\,]$	$[\,y\,]$
	`remove_maydiff` y $(a + 3)$		
	$[\,x = a + 1, y = x + 2, y = a + 3\,]$	$[\,y = a + 3\,]$	$[\ ]$
	$\{\}$	$\{\}$	$\{\}$

Figure 2: Validation of Add Assoc Optimization.

a new invariant (P', Q', D'). SIMPLEBERRY simply applies this inference rule according to the hint command. The correctness of the rule f is the following:

$$\forall \mathcal{S} : \text{State}.\ \forall P, Q, D.\ \ \mathcal{S} \models (P, Q, D) \implies \mathcal{S} \models f(P, Q, D)\ .$$

The correctness theorem of SIMPLEBERRY requires the correctness of inference rules as a precondition. Thus, one has to prove them in Coq in order to have the end-to-end formal guarantee. However, in case she is very confident about the correctness of an inference rule, she can also put the correctness into the trust base by making it as an axiom.

Overall Process The overall process of validation, according to the LLVMBERRY framework (see Figure 1), is as follows. First, the modified compiler tr′ performs an optimization and generates the core hints that consists of (i) commands for generating invariants, and (ii) commands for generating inference rules. Then, from the core hints, the hint converter conv generates invariants and inference rules as full hints, and feed them to the validator vali. Finally, the validator checks the validity of the given translation.

3 Validation Examples

In this section, we walk you through the details of SIMPLEBERRY with various examples.

3.1 Add Assoc Optimization

Our first example is the add assoc optimization. Consider the example optimization in Figure 2, where $x = a + 1$ at line 9 of block B1 followed by $y = x + 2$ at line 5 of block B4 is optimized to $x = a + 1$ followed by $y = a + 3$.

For the validation, we first need to add the following two rules, where $\textsc{Fv}(e)$ is the set of all variables occurring in e. The correctness of the rules is intuitively clear.

```
add_assoc $x$ $y$ $a$ $c_1$ $c_2$ $c_3$ in src := $\lambda(P, Q, D)$.
  if $(x = a + c_1 \in P) \wedge (y = x + c_2 \in P) \wedge c_3 = c_1 + c_2$
  then $(P \cup \{ y = a + c_3 \}, Q, D)$ else $(P, Q, D)$
remove_maydiff $x$ $e$ := $\lambda(P, Q, D)$.
  if $x = e \in P \wedge x = e \in Q \wedge \textsc{Fv}(e) \cap D = \emptyset$
  then $(P, Q, D \backslash \{ x \})$ else $(P, Q, D)$
```

Second, the modified compiler generates the core hints, consisting of invariant and inference rule commands, in Figure 2. The hint converter then generates invariants and inference rules according to the commands. The invariant command `9@B1:propagate_to 5@B4 in src` inserts the equation $x = a + 1$ at line 9 of B1 into the source property of every line between line 9 of B1 and line 5 of B4 (see the black curly bracket in Validation of Figure 2). The command `5@B4:add_assoc` x y `in src` infers the remaining arguments and inserts the inference rule `add_assoc` x y a 1 2 3 `in src` at line 5 of B4, and then `5@B4:remove_maydiff` y inserts `remove_maydiff` y $(a + 3)$ at the same line.

Finally, the validator checks the validity of the invariants. At each line, it first generates a strong post condition. For example, it generates $[\, x = a + 1, y = x + 2 \,]$, $[\, y = a + 3 \,]$, $[\, y \,]$ at line 5 of B4. It adds the invariant $y = x + 2$ in the source and $y = a + 3$ in the target because of the assignments executed. Also, it adds y to the may-diff set because it cannot simply determine whether the newly assigned values to y in the source and the target are the same. Unlike at line 5 of B4, it does not add x to the may-diff set in line 9 of B1 because the simple equality checker of Simpleberry can infer that the values in x in the source and the target are the same. Indeed, the equality checker simply checks whether the two instructions are syntactically the same and the free variables in the RHS are not in the may-diff set.

At line 9 of B1, the validator trivially checks that the strong post condition implies the invariant at the line. At line 5 of B4, it sequentially applies the `add_assoc` and `remove_maydiff` rules according to the given hints and gets the final invariant $[\, x = a + 1, y = x + 2, y = a + 3 \,]$, $[\, y = a + 3 \,]$, $[\ ]$. This invariant trivially implies $\{\ \}, \{\ \}, \{\ \}$. Thus, we checked the consistency of all invariants, which completes the validation.

3.2 Heap Optimizations

The next examples are heap optimizations: store-load optimization and dead store and alloca elimination.

Store-Load Optimization For the store-load optimization in Figure 3, we add to Simpleberry the following rule, whose correctness is easy to show.

```
trans_via_heap $x$ $y$ $p$ in src := $\lambda(P, Q, D)$.
  if $*p = x \in P \wedge *p = y \in P$
  then $(P \cup \{ x = y \}, Q, D)$ else $(P, Q, D)$
```

Then the modified compiler generates the core hints in Figure 3. The command `5@B3:propagate_to 7@B3 in src` adds the property `alc`(p) between `5@B3` and `7@B3` in the source; the command `6@B3:propagate_to 9@B3 in src` adds $*p = y$ between `6@B3` and `9@B3` in the source; and `7@B3:propagate_neq_to` q p `8@B3 in src` adds $q \neq p$ between `7@B3` and `8@B3` in the source. The command `9@B3:trans_via_heap` x y `in src` and `9@B3:remove_maydiff` x add the rules `trans_via_heap` x y p and `remove_maydiff` x y at line 9 of B3 in the source.

The validator now checks the validity of the invariants. At line 5, Simpleberry adds `alc`(p) in the post condition, but does not add p to the may-diff set because it can simply infer that p contains equivalent pointers in the source and the target that point to allocated blocks of the heaps. At line 6, SIMPLEBERRY infers that $*p = y$ holds. At line 7, it infers, in addition to `alc`(q), that $q \neq p$ holds (*i.e.,* q and p point to disjoint parts of the heap) because `alc`(p) holds in the pre condition and q is newly allocated.

At line 8, due to the heap update, SIMPLEBERRY does several things. First, it checks that the variables q and z are not in the may-diff set in order to make sure that the global heap invariant is preserved. Second, it does not remove $*p = y$ despite of heap update because there is no alias between p and q due to $q \neq p$. Finally, it adds $*q = z$ in the source and the target due to the heap update performed.

At line 9, SIMPLEBERRY adds $*p = x$ and $x = y$ in the source and the target, and then adds x to the may-diff set because it is not easy to see whether the values assigned to x are equivalent. Then, the following inference rules properly update the invariants and completes the validation, as shown in Figure 3. Here, it is important to note that since we dropped the condition `alc`(p), we do not know that p is allocated and thus the instruction $x = *p$ may fail. However, it is ok because if the source program produces an error, the target program is allowed to behave arbitrarily.

Dead Store And Alloca Elimination Thanks to the store-load optimization, line 6 and 5 became dead code and thus can be eliminated by dead store elimination followed by dead alloca elimination. This is an example that uses the isolated set invariant. For presentation purpose, we perform the two eliminations at the same time (see Figure 4).

First of all, note that we insert instructions called *logical nop* in the target program. These are a part of the hint and used only for validation purpose. Using logical nop, we can align the source and the target program.

The modified compiler generates the core hints in Figure 4. The command `*:propagate_maydiff_to` p `*` adds p to the may-diff set at every line, and `5@B3:propagate_isol_to` p `6@B3 in src` adds `isol`(p) at line 5 of B3.

SIMPLEBERRY then checks the validity of the invariants. At line 5, it treats the allocated heap in the source as an isolated heap because there is no corresponding allocation in the target, and thus adds `isol`(p) in addition to `alc`(p). At line 6, SIMPLEBERRY can make sure that the heap update in the source does not break the global heap invariant because the heap update is made to the isolated heap due to `isol`(p). The rest is straightforward.

3.3 Fold-Phi-Bin Optimization

The final example is the fold-phi-bin optimization, which optimizes a phi-node and is sensitive to the control flow of the program. In this example, we will see how logical variables are used.

A phi-node is an assignment that assigns different values depending on the incoming block. For example, in the program of Figure 5, the block B3 has two incoming blocks, B2 and B3. The instruction $\mathbf{c} = \phi(\mathbf{a@B2}, \mathbf{b@B3})$ assigns to c the value of a if coming from B2, and that of b if coming from B3.

The fold-phi-bin transformation, shown in Figure 5, defers the increment-by-one computation after the phi node, using a temporary variable t. In order to align the source and the target programs, we insert a logical nop in the source side. Since the block B3 behaves differently depending on the incoming block, SIMPLEBERRY validates B3 separately for each incoming block. In Figure 5, B3←B2 is the validation of B3 when coming from B2, and B3←B3 is the validation when coming from B3.

First, we need to add the following inference rules.

```
trans_via_var $x$ $y$ $e$ in src := $\lambda(P, Q, D)$.
  if $x = y \in P \wedge y = e \in P$
  then $(P \cup \{ x = e \}, Q, D)$ else $(P, Q, D)$
```

```
replace_var $x$ $y$ $z$ $e$ $e'$ in src := $\lambda(P, Q, D)$.
  if $x = y \in P \wedge z = e \in P \wedge e' = e[y/x]$
  then $(P \cup \{ z = e' \}, Q, D)$ else $(P, Q, D)$
intro_eq $x$ $y$ := $\lambda(P, Q, D)$.
  if $x \notin D \wedge \widehat{y} \notin \textsc{Fv}(P, Q, D)$
  then $(P \cup \{ x = \widehat{y} \}, Q \cup \{ x = \widehat{y} \}, D)$ else $(P, Q, D)$
```

Here, an interesting rule is `intro_eq`, which allows to introduce redundant equalities using a new fresh logical variable. Note that $\widehat{y} \notin \textsc{Fv}(P, Q, D)$ means that the logical variable $\widehat{y}$ does not occur in P, Q and D, *i.e.,* it is fresh. Here the newly introduced equalities say that there is some value represented by $\widehat{y}$ that is equal to the value of x in the source and the target, and furthermore the values of $\widehat{y}$ in the source and the target are equivalent because $\widehat{y} \notin D$. We will see below how such redundant equalities are used in this example.

For validation, the modified compiler generates hints that produce the invariants and inference rules shown in Figure 5. We omit how they are generated since it is similar as in the previous examples. Note that there is a single post condition of block B3 that merges the post conditions of B3←B2 and B3←B3.

The validation of B2 is straightforward. For validation of B3, Simpleberry first checks that the post condition of the block B2 implies the pre condition of block B3←B2, and the post condition of B3 implies the pre condition of B3←B3. Also it checks that the post conditions of B3←B2 and B3←B3 both imply the common post condition of block B3. In validation of B3←B2, there is nothing special except that Simpleberry adds $c = a$ and $t = x$ after execution of $\mathbf{c} = \phi(\mathbf{a@B2}, \mathbf{b@B3})$ and $\mathbf{t} = \phi(\mathbf{x@B2}, \mathbf{c@B3})$ because we assume that the incoming block is B2.

In the validation of B3←B3, at line 0 Simpleberry adds $c = b$ and $t = c$. However, in the source, we cannot simply add $c = b$ because the pre condition already contains equalities about c, whose value is not the same as the newly assigned value of c. Here, Simpleberry uses a logical variable $\widehat{c}$ to represent the old value of the variable c. It simply renames the variable c in the pre condition to $\widehat{c}$.[2] Since we assigned a value to c only in the source, Simpleberry has to add c and $\widehat{c}$ to the may-diff set. However, as a side effect, we lose the information that the value of $\widehat{c}$ in the source is equivalent to that of c in the target. It is the redundant equations $\widehat{c} = \widehat{u}$ in the source and $c = \widehat{u}$ in the target that recovers the lost information since $\widehat{u}$ is not in the may-diff set. At line 1, after renaming c to $\widehat{c}$ and adding $c = t + 1$ in the target, Simpleberry regains, via the logical variable $\widehat{u}$, the fact that c contains the equivalent value in the source and the target. Reasoning at line 2 is straightforward. Thus, the validation succeeds.

4 Semantics

Before we explain the algorithm and the verification, we first present a formal semantics. For the presentation purpose, we abstract away the unnecessary details of LLVM IR. In other words, we present a semantics for an idealized LLVM IR.

4.1 Program Semantics

$\mathcal{C} = (G, F) \in$ Config is a static configuration initialized after loading a program $\mathcal{P}$, where $G \in \text{Id} \rightarrow$ `option` Addr is a global variable map that maps a variable name to the associated heap address, and $F \in \text{Addr} \rightarrow$ `option` Fdef is a function lookup table that maps a function pointer to its declaration and definition. Here, `option` T is the usual option type consisting of either $\bot$ or $\lfloor t \rfloor$ for some $t \in T$. A function name is also a global variable, so that you can

[2]Here, before renaming c to $\widehat{c}$, Simpleberry first removes every predicate containing $\widehat{c}$ from the pre condition. Though there will not be any such predicate because the program is in SSA, this removal makes the verification of Simpleberry easier.

get the definition of a function, say `foo`, by first looking up `foo` in G (*i.e.,* $\lfloor p \rfloor = G(\texttt{foo})$) and then get the definition via F (*i.e.,* $\lfloor \mathrm{fd} \rfloor = F(p)$). Note that the static configuration $\mathcal{C}$ remains the same during the execution. Throughout this section, by $\mathcal{C}_{\mathrm{src}}$ and $\mathcal{C}_{\mathrm{tgt}}$ we denote the static configurations of the source and the target program.

$\mathcal{S} = (\sigma, h) \in$ State is a *program state* that changes over time during the execution, and consists of a *call stack* $\sigma \in \Sigma(=$ list $\mathbb{F})$ and a *heap* $h \in$ Heap$(=$ Addr $\rightharpoonup$ Value$\uplus\{$dead$\})$. Here, a call stack σ is a list of stack frames and a heap h is a partial function that maps heap addresses to either values or dead cells (dead). An allocated cell becomes dead when it is freed, and is never used for allocation again. A *stack frame* $(pc, f, s) \in \mathbb{F}$ consists of a program counter *pc*, the current function f, and the local store $s \in$ Store$(=$ Id $\to$ Value$)$. A program counter $pc(= n@B_i)$, consisting of a block label B_i and a line number n, indicates the current code pointer being executed inside f. The local store s maps variables to values. A value $v \in$ Value$(=$ Int$\uplus$Addr$)$ is either an integer, or an address (pointer value).

The small-step operational semantics is a judgment of the form:

$$\mathcal{C} \vdash \mathcal{S} \to^{\varepsilon} \mathcal{S}' \ ,$$

which denotes that in the static configuration $\mathcal{C}$, the current state $\mathcal{S}$ steps to a next state $\mathcal{S}'$, emitting an observable IO event ε. You can understand this as a usual small-step semantics.

As we have seen in the previous examples, we need to align instructions using logical nops. We can easily derive an instrumented semantics that takes into account a hint N for the locations of logical nops, which has the following form:

$$(\mathcal{C}, N) \vdash \mathcal{S} \to^{\varepsilon} \mathcal{S}' \ .$$

4.2 Hint Semantics

As we have seen in Section 2, we have two kinds of invariants: the implicit heap invariant and invariants at each line of code.

Implicit Heap Invariant As discussed in Section 2, the implicit heap invariant relating the source heap h_{src} and the target heap h_{tgt} partitions them into public and isolated heaps ($h^{\mathrm{e}}_{\mathrm{src}} \uplus h^{\mathrm{i}}_{\mathrm{src}} \subseteq h_{\mathrm{src}}$ and $h^{\mathrm{e}}_{\mathrm{tgt}} \uplus h^{\mathrm{i}}_{\mathrm{tgt}} \subseteq h_{\mathrm{tgt}}$). To represent such an invariant we need partitioning information α[3], β_{src}, and β_{tgt}, where α is a bijection between the addresses of $h^{\mathrm{e}}_{\mathrm{src}}$ and $h^{\mathrm{e}}_{\mathrm{tgt}}$, β_{src} is the set of addresses in $h^{\mathrm{i}}_{\mathrm{src}}$, and similarly for β_{tgt}. Of course, β_{src} should be disjoint from *alpha*, and the same for β_{tgt}.

In addition to the partitioning, we need more properties on the heaps.

- Every cell in the isolated heaps should not be dead:

$$(\forall a \in \beta_{\mathrm{src}}.h_{\mathrm{src}}(a) \neq \mathrm{dead}) \land$$
$$(\forall a \in \beta_{\mathrm{tgt}}, h_{\mathrm{tgt}}(a) \neq \mathrm{dead}) \ .$$

- The values in the corresponding locations of the public heaps w.r.t. α should be equivalent modulo α, denoted $h^{\mathrm{e}}_{\mathrm{src}} \overset{\alpha}{\simeq} h^{\mathrm{e}}_{\mathrm{tgt}}$, if: $\forall (a_{\mathrm{src}}, a_{\mathrm{tgt}}) \in \alpha$.

$$(h^{\mathrm{e}}_{\mathrm{src}}(a_{\mathrm{src}}) = h^{\mathrm{e}}_{\mathrm{tgt}}(a_{\mathrm{tgt}}) = \mathrm{dead}) \ \lor (h^{\mathrm{e}}_{\mathrm{src}}(a_{\mathrm{src}}) \overset{\alpha}{\simeq} h^{\mathrm{e}}_{\mathrm{tgt}}(a_{\mathrm{tgt}})) \ ,$$

where values v_{src} and v_{tgt} are equivalent modulo α, denoted $v_{\mathrm{src}} \overset{\alpha}{\simeq} v_{\mathrm{tgt}}$, if:

$$(v_{\mathrm{src}} = v_{\mathrm{tgt}} \in \mathrm{Int}) \ \lor \ ((v_{\mathrm{src}}, v_{\mathrm{tgt}}) \in \alpha) \ .$$

We write $(h_{\mathrm{src}}, h_{\mathrm{tgt}}) \in \mathcal{H}(\alpha, \beta_{\mathrm{src}}, \beta_{\mathrm{tgt}})$ when the heap invariant holds for h_{src} and h_{tgt}.

[3]The notion of memory injection α is originally used in the verification of CompCert [5].

Transformation

B3		
5:	$\mathbf{p = alloca(1)}$	$\mapsto \mathbf{p = alloca(1)}$
6:	$\mathbf{*p = y}$	$\mapsto \mathbf{*p = y}$
7:	$\mathbf{q = alloca(1)}$	$\mapsto \mathbf{q = alloca(1)}$
8:	$\mathbf{*q = z}$	$\mapsto \mathbf{*q = z}$
9:	$\mathbf{x = *p}$	$\mapsto \mathbf{x = y}$

Invariant Commands

5@B3: `propagate_to 7@B3 in src`
6@B3: `propagate_to 9@B3 in src`
7@B3: `propagate_neq_to` q p `8@B3 in src`

Inference Rule Commands

9@B3: `trans_via_heap` x y `in src`
9@B3: `remove_maydiff` x

Validation

B3			
$\{\ \}$		$\{\ \}$	$\{\ \}$
5: $\mathbf{p = alloca(1)}$	$\mapsto$	$\mathbf{p = alloca(1)}$	
$[\,\texttt{alc}(p)\,]$		$[\,\texttt{alc}(p)\,]$	$[\ ]$
$\{\,\texttt{alc}(p)\,\}$		$\{\ \}$	$\{\ \}$
6: $\mathbf{*p = y}$	$\mapsto$	$\mathbf{*p = y}$	
$[\,\texttt{alc}(p), *p = y\,]$		$[\,*p = y\,]$	$[\ ]$
$\{\,\texttt{alc}(p), *p = y\,\}$		$\{\ \}$	$\{\ \}$
7: $\mathbf{q = alloca(1)}$	$\mapsto$	$\mathbf{q = alloca(1)}$	
$[\,\texttt{alc}(p), *p = y, \texttt{alc}(q), q \neq p\,]$		$[\,\texttt{alc}(q)\,]$	$[\ ]$
$\{\,*p = y, q \neq p\,\}$		$\{\ \}$	$\{\ \}$
8: $\mathbf{*q = z}$	$\mapsto$	$\mathbf{*q = z}$	
$[\,*p = y, q \neq p, *q = z\,]$		$[\,*q = z\,]$	$[\ ]$
$\{\,*p = y\,\}$		$\{\ \}$	$\{\ \}$
9: $\mathbf{x = *p}$	$\mapsto$	$\mathbf{x = y}$	
$[\,*p = y, *p = x\,]$		$[\,x = y\,]$	$[\,x\,]$
`trans_via_heap` x y p `in src`			
$[\,*p = y, *p = x, x = y\,]$		$[\,x = y\,]$	$[\,x\,]$
`remove_maydiff` x y			
$[\,*p = y, *p = x, x = y\,]$		$[\,x = y\,]$	$[\ ]$
$\{\ \}$		$\{\ \}$	$\{\ \}$

Figure 3: Validation of Store-Load Optimization.

Transformation

B3		
5: **p = alloca(1)**	↦	**logicalnop**
6: ∗**p = y**	↦	**logicalnop**
7: **q = alloca(1)**	↦	**q = alloca(1)**
8: ∗**q = z**	↦	∗**q = z**
9: **x = y**	↦	**x = y**

Invariant Commands

```
*: propagate_maydiff_to
  p *
5@B3: propagate_isol_to
      p 6@B3 in src
```

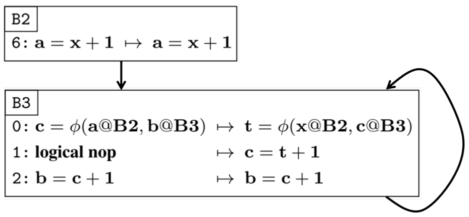
Inference Rule Commands

Validation

B3			
{ }		{ }	$\{\, p \,\}$
5: **p = alloca(1)**	↦	**logical nop**	
[alc(p), isol(p)]		[ ]	$[\, p \,]$
{ isol(p) }		{ }	$\{\, p \,\}$
6: ∗**p = y**	↦	**logical nop**	
[isol(p), $*p = y$]		[ ]	$[\, p \,]$
{ }		{ }	$\{\, p \,\}$
7: **q = alloca(1)**	↦	**q = alloca(1)**	
[alc(q)]		[alc(q)]	$[\, p \,]$
{ }		{ }	$\{\, p \,\}$
8: ∗**q = z**	↦	∗**q = z**	
[$*q = z$]		[$*q = z$]	$[\, p \,]$
{ }		{ }	$\{\, p \,\}$
9: **x = y**	↦	**x = y**	
[$x = y$]		[$x = y$]	$[\, p \,]$
{ }		{ }	$\{\, p \,\}$

Figure 4: Validation of Dead Store & Alloca Elimination.

Transformation


B2
6: a = x + 1 ↦ a = x + 1
B3
0: c = φ(a@B2, b@B3) ↦ t = φ(x@B2, c@B3)
1: logical nop ↦ c = t + 1
2: b = c + 1 ↦ b = c + 1

Validation

B2			
	$\{\ \}$	$\{\ \}$	$\{t, \widehat{c}\}$
$6:\ \mathbf{a = x + 1}$	$\mapsto$	$\mathbf{a = x + 1}$	
	$[\,a = x + 1\,]$	$[\,a = x + 1\,]$	$[\,t, \widehat{c}\,]$
	$\{\,a = x + 1\,\}$	$\{\ \}$	$\{\,t, \widehat{c}\,\}$

B3←B2		
$\{a = x{+}1\}$	$\{\}$	$\{t, \widehat{c}\}$
$0{:}\mathbf{c} = \phi(\mathbf{a@B2}, \mathbf{b@B3})$	$\mapsto \mathbf{t} = \phi(\mathbf{x@B2}, \mathbf{c@B3})$	
$[a = x{+}1, c = a]$	$[t = x]$	$[t, \widehat{c}, c]$
$\{a = x{+}1, c = a\}$	$\{t = x\}$	$\{t, \widehat{c}, c\}$
$1{:}$**logical nop**	$\mapsto \mathbf{c = t + 1}$	
$[a{=}x{+}1, c{=}a]$	$[t{=}x, c{=}t{+}1]$	$[t, \widehat{c}, c]$
`trans_via_var` $c\ a\ (x+1)$ `in src`		
$[a{=}x{+}1, c{=}a, c{=}x{+}1]$	$[t{=}x, c{=}t{+}1]$	$[t, \widehat{c}, c]$
`replace_var` $t\ x\ c\ (t+1)\ (x+1)$ `in tgt`		
$[a{=}x{+}1, c{=}a, c{=}x{+}1]$	$[t{=}x, c{=}t{+}1, c{=}x{+}1]$	$[t, \widehat{c}, c]$
`remove_maydiff` $c\ (x+1)$		
$[a{=}x{+}1, c{=}a, c{=}x{+}1]$	$[t{=}x, c{=}t{+}1, c{=}x{+}1]$	$[t, \widehat{c}]$
$\{\}$	$\{\}$	$\{t, \widehat{c}\}$
$2{:}\mathbf{b = c + 1}$	$\mapsto \mathbf{b = c + 1}$	
$[b = c + 1]$	$[b = c + 1]$	$[t, \widehat{c}]$
`intro_eq` $c\ u$		
$[b = c + 1, c = \widehat{u}]$	$[b = c + 1, c = \widehat{u}]$	$[t, \widehat{c}]$
$\{b = c{+}1, c = \widehat{u}\}$	$\{c = \widehat{u}\}$	$\{t, \widehat{c}\}$

B3←B3		
$\{b = c{+}1, c = \widehat{u}\}$	$\{c = \widehat{u}\}$	$\{t, \widehat{c}\}$
$0{:}\mathbf{c} = \phi(\mathbf{a@B2}, \mathbf{b@B3})$	$\mapsto \mathbf{t} = \phi(\mathbf{x@B2}, \mathbf{c@B3})$	
$[b = \widehat{c}{+}1, \widehat{c} = \widehat{u}, c = b]$	$[c = \widehat{u}, t = c]$	$[t, \widehat{c}, c]$
$\{b = \widehat{c}{+}1, \widehat{c} = \widehat{u}, c = b\}$	$\{c = \widehat{u}, t = c\}$	$\{t, \widehat{c}, c\}$
$1{:}$**logical nop**	$\mapsto \mathbf{c = t + 1}$	
$[b{=}\widehat{c}{+}1, \widehat{c}{=}\widehat{u}, c{=}b]$	$[\widehat{c}{=}\widehat{u}, t{=}\widehat{c}, c{=}t{+}1]$	$[t, \widehat{c}, c]$
`trans_via_var` $c\ b\ (\widehat{c}+1)$ `in src`		
$[b{=}\widehat{c}{+}1, \widehat{c}{=}\widehat{u}, c{=}b, c{=}\widehat{c}{+}1]$	$[\widehat{c}{=}\widehat{u}, t{=}\widehat{c}, c{=}t{+}1]$	$[t, \widehat{c}, c]$
`replace_var` $\widehat{c}\ \widehat{u}\ c\ (\widehat{c}+1)\ (\widehat{u}+1)$ `in src`		
$[b{=}\widehat{c}{+}1, \widehat{c}{=}\widehat{u}, c{=}b, c{=}\widehat{c}{+}1, c{=}\widehat{u}{+}1]$	$[\widehat{c}{=}\widehat{u}, t{=}\widehat{c}, c{=}t{+}1]$	$[t, \widehat{c}, c]$
`trans_via_var` $t\ \widehat{c}\ \widehat{u}$ `in tgt`		
$[b{=}\widehat{c}{+}1, \widehat{c}{=}\widehat{u}, c{=}b, c{=}\widehat{c}{+}1, c{=}\widehat{u}{+}1]$	$[\widehat{c}{=}\widehat{u}, t{=}\widehat{c}, c{=}t{+}1, c{=}\widehat{u}]$	$[t, \widehat{c}, c]$
`replace_var` $t\ \widehat{u}\ c\ (t+1)\ (\widehat{u}+1)$ `in tgt`		
$[b{=}\widehat{c}{+}1, \widehat{c}{=}\widehat{u}, c{=}b, c{=}\widehat{c}{+}1, c{=}\widehat{u}{+}1]$	$[\widehat{c}{=}\widehat{u}, t{=}\widehat{c}, c{=}t{+}1, c{=}\widehat{u}, c{=}\widehat{u}{+}1]$	$[t, \widehat{c}, c]$
`remove_maydiff` $c\ (\widehat{u}+1)$		
$[b{=}\widehat{c}{+}1, \widehat{c}{=}\widehat{u}, c{=}b, c{=}\widehat{c}{+}1, c{=}\widehat{u}{+}1]$	$[\widehat{c}{=}\widehat{u}, t{=}\widehat{c}, c{=}t{+}1, c{=}\widehat{u}, c{=}\widehat{u}{+}1]$	$[t, \widehat{c}]$
$\{\}$	$\{\}$	$\{t, \widehat{c}\}$
$2{:}\mathbf{b = c + 1}$	$\mapsto \mathbf{b = c + 1}$	
$[b = c + 1]$	$[b = c + 1]$	$[t, \widehat{c}]$
`intro_eq` $c\ u$		
$[b = c + 1, c = \widehat{u}]$	$[b = c + 1, c = \widehat{u}]$	$[t, \widehat{c}]$
$\{b = c{+}1, c = \widehat{u}\}$	$\{c = \widehat{u}\}$	$\{t, \widehat{c}\}$

$\{b = c{+}1, c = \widehat{u}\}$	$\{c = \widehat{u}\}$	$\{t, \widehat{c}\}$

Figure 5: Validation of Fold-Phi-Bin Optimization.

```
**vali** $\mathcal{P}_{\text{src}}\ \mathcal{P}_{\text{tgt}}\ N\ R\ (P, Q, D) :=$
    1. check $\mathcal{P}_{\text{src}} = \mathcal{P}_{\text{tgt}}$ except for the code of each function.
    2. check, for every function $f$,
       **vali_fun** $\mathcal{P}_{\text{src}}\ \mathcal{P}_{\text{tgt}}\ f\ N(f)\ R(f)\ (P(f), Q(f), D(f)) =$ true
```

```
**postcond** $\mathcal{P}_{\text{src}}\ \mathcal{P}_{\text{tgt}}\ i_{\text{src}}\ i_{\text{tgt}}\ (P, Q, D) :=$
$(\hat{s}_{\text{src}}, \hat{s}_{\text{tgt}}, s_{\text{src}}, s_{\text{tgt}}, h_{\text{src}}, h_{\text{tgt}}) \in [\![(P, Q, D)]\!]\ (\alpha, \beta_{\text{src}}, \beta_{\text{tgt}})$
$\wedge\ (h_{\text{src}}, h_{\text{tgt}}) \in \mathcal{H}(\alpha, \beta_{\text{src}}, \beta_{\text{tgt}})$
    1. check $i_{\text{src}}$ and $i_{\text{tgt}}$ are consistent.
Construct $\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}}$ such that $\alpha \sqsubseteq \alpha'$
$\wedge\ (\hat{s}_{\text{src}}, \hat{s}_{\text{tgt}}, s_{\text{src}}, s_{\text{tgt}}, h_{\text{src}}, h_{\text{tgt}}) \in [\![(P, Q, D)]\!]\ (\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
$\wedge\ (h'_{\text{src}}, h'_{\text{tgt}}) \in \mathcal{H}(\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
    2. let $(P', Q', D') =$ rename_to_log_var_add_md $i_{\text{src}}\ i_{\text{tgt}}\ (P, Q, D)$.
Construct $\hat{s}'_{\text{src}}, \hat{s}'_{\text{tgt}}$ such that
$(\hat{s}'_{\text{src}}, \hat{s}'_{\text{tgt}}, s'_{\text{src}}, s'_{\text{tgt}}, h_{\text{src}}, h_{\text{tgt}}) \in [\![(P', Q', D')]\!]\ (\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
$\wedge\ (h'_{\text{src}}, h'_{\text{tgt}}) \in \mathcal{H}(\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
    3. let $(P'', Q'', D'') =$ del_heap_prop $i_{\text{src}}\ i_{\text{tgt}}\ (P', Q', D')$.
$(\hat{s}'_{\text{src}}, \hat{s}'_{\text{tgt}}, s'_{\text{src}}, s'_{\text{tgt}}, h'_{\text{src}}, h'_{\text{tgt}}) \in [\![(P'', Q'', D'')]\!]\ (\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
$\wedge\ (h'_{\text{src}}, h'_{\text{tgt}}) \in \mathcal{H}(\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
    4. let $(P''', Q''', D''') =$ del_md_add_prop $i_{\text{src}}\ i_{\text{tgt}}\ (P'', Q'', D'')$.
$(\hat{s}'_{\text{src}}, \hat{s}'_{\text{tgt}}, s'_{\text{src}}, s'_{\text{tgt}}, h'_{\text{src}}, h'_{\text{tgt}}) \in [\![(P''', Q''', D''')]\!]\ (\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
$\wedge\ (h'_{\text{src}}, h'_{\text{tgt}}) \in \mathcal{H}(\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
    5. return $(P''', Q''', D''')$
```

```
**vali_fun** $\mathcal{P}_{\text{src}}\ \mathcal{P}_{\text{tgt}}\ f\ N\ R\ (P, Q, D) :=$
    1. check $\text{CFG}(\mathcal{P}_{\text{src}}(f)) = \text{CFG}(\mathcal{P}_{\text{tgt}}(f))$.
    2. check $P_{\text{pre}@B_{\text{init}}} = \emptyset \wedge Q_{\text{pre}@B_{\text{init}}} = \emptyset$.
    3. For every branch $B_i \to B_j \in \text{CFG}(\mathcal{P}_{\text{src}}(f))$,
       check $P_{\text{post}@B_i} \supseteq P_{\text{pre}@B_j \leftarrow B_i} \wedge Q_{\text{post}@B_i} \supseteq$
       $Q_{\text{pre}@B_j \leftarrow B_i} \wedge D_{\text{post}@B_i} \subseteq D_{\text{pre}@B_j \leftarrow B_i}$.
    4. For each line $n$ of each block $B_i$, check the
       following.
        (a) let $(i_{\text{src}}, i_{\text{tgt}}) =$ get_instrs $\mathcal{P}_{\text{src}}\ \mathcal{P}_{\text{tgt}}\ N\ n$.
       $(\hat{s}_{\text{src}}, \hat{s}_{\text{tgt}}, s_{\text{src}}, s_{\text{tgt}}, h_{\text{src}}, h_{\text{tgt}}) \in [\![(P_{n-1}, Q_{n-1}, D_{n-1})]\!]\ (\alpha, \beta_{\text{src}}, \beta_{\text{tgt}})$
       $\wedge\ (h_{\text{src}}, h_{\text{tgt}}) \in \mathcal{H}(\alpha, \beta_{\text{src}}, \beta_{\text{tgt}})$
        (b) let $(P', Q', D') =$
                **postcond** $\mathcal{P}_{\text{src}}\ \mathcal{P}_{\text{tgt}}\ i_{\text{src}}\ i_{\text{tgt}}\ (P_{n-1}, Q_{n-1}, D_{n-1})$.
       Construct $\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}}, \hat{s}'_{\text{src}}, \hat{s}'_{\text{tgt}}$ such that $\alpha \sqsubseteq \alpha'$
       $\wedge\ (\hat{s}'_{\text{src}}, \hat{s}'_{\text{tgt}}, s'_{\text{src}}, s'_{\text{tgt}}, h'_{\text{src}}, h'_{\text{tgt}}) \in [\![(P', Q', D')]\!]\ (\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
       $\wedge\ (h'_{\text{src}}, h'_{\text{tgt}}) \in \mathcal{H}(\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
        (c) let $(P'', Q'', D'') =$
            apply_inf_rules $R\ (P', Q', D')$.
       $(\hat{s}'_{\text{src}}, \hat{s}'_{\text{tgt}}, s'_{\text{src}}, s'_{\text{tgt}}, h'_{\text{src}}, h'_{\text{tgt}}) \in [\![(P'', Q'', D'')]\!]\ (\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
       $\wedge\ (h'_{\text{src}}, h'_{\text{tgt}}) \in \mathcal{H}(\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
        (d) check $P'' \supseteq P_n \wedge Q'' \supseteq Q_n \wedge D'' \subseteq D_n$.
       $(\hat{s}'_{\text{src}}, \hat{s}'_{\text{tgt}}, s'_{\text{src}}, s'_{\text{tgt}}, h'_{\text{src}}, h'_{\text{tgt}}) \in [\![(P_n, Q_n, D_n)]\!]\ (\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
       $\wedge\ (h'_{\text{src}}, h'_{\text{tgt}}) \in \mathcal{H}(\alpha', \beta'_{\text{src}}, \beta'_{\text{tgt}})$
```

Figure 6: Algorithm and Specification of SIMPLEBERRY.

Unary Invariant A unary invariant is a property P for the source or Q for the target program. What we define here applies to both the source and the target programs. The type of $[\![P]\!]$ is as follows:

$$[\![P]\!](\beta) \subseteq \text{Store} \times \text{Store} \times \text{Heap}$$

The semantics of P relies on isolated addresses β and relates the logical store $\hat{s}$, the physical store s, and the heap h, where the logical store is not connected to the physical state, and just maps logical variables to values.

The semantics of P is the conjunction of that of each element in P:

$$(\hat{s}, s, h) \in [\![P]\!](\beta) \iff \forall p \in P.\ (\hat{s}, s, h) \in [\![p]\!](\beta)\ .$$

The semantics of each basic property is given as follows:

- $[\![x = e]\!](\beta) = \{(\hat{s}, s, h) \mid [\![x]\!]_{\hat{s},s} = [\![e]\!]_{\hat{s},s}\}$
- $[\![*p = x]\!](\beta) = \{(\hat{s}, s, h) \mid [\![x]\!]_{\hat{s},s} = h([\![p]\!]_{\hat{s},s})\}$
- $[\![p \neq q]\!](\beta) = \{(\hat{s}, s, h) \mid [\![p]\!]_{\hat{s},s} \neq [\![q]\!]_{\hat{s},s}\}$
- $[\![\texttt{alc}(p)]\!](\beta) = \{(\hat{s}, s, h) \mid \exists v.\ h([\![p]\!]_{\hat{s},s}) = v\}$
- $[\![\texttt{isol}(p)]\!](\beta) = \{(\hat{s}, s, h) \mid [\![p]\!]_{\hat{s},s} \in \beta\}$,

where the variable x can be either a physical or a logical variable. $[\![x]\!]_{\hat{s},s}$ looks up the logical store $\hat{s}$ or the physical store s depending on whether x is logical or physical.The semantics $[\![e]\!]_{\hat{s},s}$ of expression is structurally defined as usual. The rest of the semantics is straightforward.

May-Diff Set The semantics of a may-diff set D relies on the memory injection α and relates the source and target stores.

$$\begin{aligned}[\![D]\!](\alpha) := \{(\hat{s}_{\text{src}}, \hat{s}_{\text{tgt}}, s_{\text{src}}, s_{\text{tgt}}) \mid \\ \forall x \notin D.\ [\![x]\!]_{\hat{s}_{\text{src}},s_{\text{src}}} \overset{\alpha}{\simeq} [\![x]\!]_{\hat{s}_{\text{tgt}},s_{\text{tgt}}}\}\ .\end{aligned}$$

It simply says that every variable not in D should be equivalent modulo α.

We say that the logical and physical stores and heaps in the source and the target satisfy an invariant (P, Q, D), denoted:

$$(\hat{s}_{\text{src}}, \hat{s}_{\text{tgt}}, s_{\text{src}}, s_{\text{tgt}}, h_{\text{src}}, h_{\text{tgt}}) \in [\![(P, Q, D)]\!](\alpha, \beta_{\text{src}}, \beta_{\text{tgt}})\ ,$$

if they satisfy all the semantics of P, Q, and D.

5 Algorithm and Verification of Simpleberry

5.1 Algorithm

In this section, we explain the high-level algorithm of Simpleberry, given in Figure 6. Ignore the specifications of the algorithm written in blue color for now.

Given the source and the target programs $\mathcal{P}_{\text{src}}, \mathcal{P}_{\text{tgt}}$ and hints N for logical nops, R for inference rules, and (P, Q, D) for invariants, the validator **vali** checks two things. It first checks that the optimization only changed the code of each function, and then checks that the translation made in each function is valid.

The function validator **vali_fun** takes a function name f as argument and locally checks the validity of the translation in f. It first checks that the control flow graphs of the function

f in the source and the target are equal. Second, it checks that the pre condition of the initial block is maximally permissive (*i.e.,* $P, Q = \emptyset$ and D can be arbitrary). Third, it checks that whenever there is a branch from B_i to B_j, the post condition of B_i implies the corresponding pre condition of B_j. Finally, it validates each line of the function code.

At each line n of block B_i, SIMPLEBERRY first obtains the current instructions i_{src} and i_{tgt} in the source and the target programs. Here, it uses the logical nop hint N and the obtained instruction may be the logical nop. Second, it calculates a post condition of the current instructions. Third, it applies the inference rules R to the post condition. Finally, it checks that the derived invariant implies the invariant at the next line.

Finally, we explain the post condition calculator **postcond** in more detail. First of all, it checks that the current instructions are consistent; otherwise, it fails at validation. Examples of inconsistency include violation of the implicit heap invariant, or raise of different side effects such as error or function call. Second, it renames a variable x to the logical variable $\widehat{x}$ in the invariant if the variable x gets updated by the current instruction. Moreover, it adds both x and $\widehat{x}$ to the may-diff set.

Third, it removes from the invariant the heap properties that may not hold any more. For example, if an unknown part of the heap is updated or a non-read only function is called, we drop all heap equalities of the form $*p = x$ from the invariant. Finally, we remove from the may-diff set the variable updated with equivalent values in the source and the target, and add new properties that hold for the new state after executing the current instruction.

5.2 Verification of Core Modules

In this section, we discuss how we verify the core modules of SIMPLEBERRY: **vali_fun** and **postcond**.

First of all, since the source and the target programs have the same control flow graph by item 1 of **vali_fun**, we can validate the translation block by block. Furthermore, since we align the instructions of the source and the target programs by the nop hint N, we can validate the translation line by line.

What we need to show for each function f is three fold. First, we need to show that whenever a function f is invoked in the source and the target programs, the program states satisfy the pre condition of the function (*i.e.,* $P_{\text{pre}@B_{\text{init}}}, Q_{\text{pre}@B_{\text{init}}}, D_{\text{pre}@B_{\text{init}}}$). This follows from item 2 of **vali_fun** because $P_{\text{pre}@B_{\text{init}}} = \emptyset$, $Q_{\text{pre}@B_{\text{init}}} = \emptyset$ and the local variables of each function can be assumed to initially have equivalent values in the source and the target. Second, whenever jumping to a new block, the program states satisfy the pre condition of the block. This follows from item 3 of **vali_fun** trivially. Finally, we show that at each line of a block, if the program states satisfy the pre condition of the line, then after execution of the instructions (or function calls), the updated program states satisfy the post condition of the line. This last step is the main part of SIMPLEBERRY and we give the detailed specifications of the algorithm in blue color in Figure 6 .

We walk through the algorithm and explain the high-level ideas of the verification. First at 4-(b), we assume that the current store and heaps $s_{\text{src}}, h_{\text{src}}, s_{\text{tgt}}, h_{\text{tgt}}$ of the source and the target programs satisfy the implicit heap invariant $\mathcal{H}$ and the pre condition $(P_{n-1}, Q_{n-1}, D_{n-1})$ for some partitioning $\alpha, \beta_{\text{src}}, \beta_{\text{tgt}}$ and logical stores $\hat{s}_{\text{src}}, \hat{s}_{\text{tgt}}$. Then the function **postcond** calculates a post condition (P', Q', D') that is satisfied by the program states $s'_{\text{src}}, h'_{\text{src}}, s'_{\text{tgt}}, h'_{\text{tgt}}$ after execution. Also, if **postcond** succeeds, the heap invariant $\mathcal{H}$ is guaranteed to hold. At 4-(c), by the correctness of inference rules, we have that the derived invariant (P'', Q'', D'') holds. By 4-(d), we can easily see that the post condition (P_n, Q_n, D_n) also holds.

The verification of **postcond** is the most interesting. Before item 1 of **postcond**, we assume that the current state satisfies (P, Q, D) and the heap invariant $\mathcal{H}$ for some partitioning and logical stores. We now walk through each step of **postcond**.

1. We check that the instructions executed or the functions called preserve the heap invariant

$\mathcal{H}$. For example, when $i_{\mathrm{src}} = (\mathbf{p}{=}\mathbf{alloca}(\mathbf{x}))$ and $i_{\mathrm{tgt}} = (\mathbf{q}{=}\mathbf{alloca}(\mathbf{y}))$, Simpleberry checks that the size of allocation is the same by checking for some variable $z \notin D$ that the source has equality $x = z$ and the target has $y = z$. Then, we can construct

$$\alpha' = \alpha \uplus (b_{\mathrm{src}}, b_{\mathrm{tgt}}),\ \beta'_{\mathrm{src}} = \beta_{\mathrm{src}},\ \beta'_{\mathrm{tgt}} = \beta_{\mathrm{tgt}}$$

for $b_{\mathrm{src}}, b_{\mathrm{tgt}}$ the newly allocated blocks in the source and the target. It is easy to see that the new states satisfy the heap invariant $\mathcal{H}(\alpha', \beta'_{\mathrm{src}}, \beta'_{\mathrm{tgt}})$. In the function call case, as long as we show that the same functions are called with the same arguments in the source and the target, we can freely assume that the returned states satisfy the heap invariant $\mathcal{H}$ for some extended α'.

2. Since we rename a newly defined variable x to the logical variable $\widehat{x}$, we have to move the old value of x to the logical store. By renaming the equivalence between the source and the target may not hold any more, we add the renamed variables to the may-diff set. For example, if the pre condition has equality $x = 5$ and $i_{\mathrm{src}} = (\mathbf{x}{=}\mathbf{1})$, then we construct $\hat{s}'_{\mathrm{src}} = \hat{s}_{\mathrm{src}}[x \leftarrow s_{\mathrm{src}}(x)]$, so that the renamed equality $\hat{x} = 5$ holds. Note that since P', Q' still contains the heap equations for the old heaps h_{src} and h_{tgt}, we still use the old heaps in the specification.

3. Since del_heap_prop removes all heap equations that may not hold any more in the updated heaps, we can show that the new heaps h'_{src} and h'_{tgt} satisfy (P'', Q'', D''). For example, if we call non-readonly functions, they may change the heap completely, so that Simpleberry drops all heap equations from the invariant.

4. Finally, we show that the newly added equations hold for the new state and the variables dropped from the may-diff set are equivalent in the new states.

5.3 Simulation

We show the final goal of program refinement between the source and the target programs (*i.e.,* the behavior of the target program is included in that of the source) using the simulation relation technique. To this end, we first define a relation $\sim$ for each function that relates the source and the target programs, and show that the relation $\sim$ is a simulation relation w.r.t. the instrumented operational semantics (*i.e.,* taking into account the logical nops). From this, by a standard argument, it follows that the target refines the source w.r.t. the instrumented semantics, which in turn implies that the same holds w.r.t. the real semantics by a simple coinduction.

Local Simulation Relation Suppose that the validation succeeds (*i.e.,* **vali** $\mathcal{P}_{\mathrm{src}}\, \mathcal{P}_{\mathrm{tgt}}\, N\, R\, (P, Q, D) =$ `true`). Let $\mathcal{C}_{\mathrm{src}}, \mathcal{C}_{\mathrm{tgt}}$ be the static configurations of $\mathcal{P}_{\mathrm{src}}$ and $\mathcal{P}_{\mathrm{tgt}}$ after loading. Let f be any function of the program, ps_{src}, ps_{tgt} be any parent stack frames. Then we define a relation $\sim$ as follows:

$$\begin{aligned}&\mathcal{S}_{\mathrm{src}} \overset{\alpha}{\sim} \mathcal{S}_{\mathrm{tgt}} \iff \exists \hat{s}_{\mathrm{src}}, \hat{s}_{\mathrm{tgt}}, \beta_{\mathrm{src}}, \beta_{\mathrm{tgt}}.\\ &\quad (h_{\mathrm{src}}, h_{\mathrm{tgt}}) \in \mathcal{H}(\alpha, \beta_{\mathrm{src}}, \beta_{\mathrm{tgt}}) \wedge \\ &\quad (\hat{s}_{\mathrm{src}}, \hat{s}_{\mathrm{tgt}}, s_{\mathrm{src}}, s_{\mathrm{tgt}}, h_{\mathrm{src}}, h_{\mathrm{tgt}}) \in [\![(P_{pc}, Q_{pc}, D_{pc})]\!]\,(\alpha, \beta_{\mathrm{src}}, \beta_{\mathrm{tgt}})\ .\end{aligned}$$

where $\mathcal{S}_{\mathrm{src}} = ((pc, f, s_{\mathrm{src}}) :: ps_{\mathrm{src}}, h_{\mathrm{src}})$ and $\mathcal{S}_{\mathrm{tgt}} = ((pc, f, s_{\mathrm{tgt}}) :: ps_{\mathrm{tgt}}, h_{\mathrm{tgt}})$. Here we simply relates any program states with the same program counter of the same function that satisfy the invariant at the current program counter pc. Of course, the initial states of the source and the target programs after loading are related by $\sim$ for some α because the pre condition of the initial block is empty.

Then using the verification results of **vali** in the previous section, we can show that the relation $\sim$ is a simulation relation in the following sense.

Theorem 1 (Simulation)**.** For any α and $\mathcal{S}_{\mathrm{src}} \overset{\alpha}{\sim} \mathcal{S}_{\mathrm{tgt}}$, we have:
$\forall \mathcal{S}'_{\mathrm{src}} \forall \varepsilon.\ (\mathcal{C}_{\mathrm{src}}, N_{\mathrm{src}}) \vdash \mathcal{S}_{\mathrm{src}} \to^{\varepsilon} \mathcal{S}'_{\mathrm{src}} \implies \exists \mathcal{S}'_{\mathrm{tgt}}.\ (\mathcal{C}_{\mathrm{tgt}}, N_{\mathrm{tgt}}) \vdash \mathcal{S}_{\mathrm{tgt}} \to^{\varepsilon} \mathcal{S}'_{\mathrm{tgt}} \wedge$

1. $(\forall \mathcal{S}'_{\mathrm{tgt}} \forall \varepsilon.\ (\mathcal{C}_{\mathrm{tgt}}, N_{\mathrm{tgt}}) \vdash \mathcal{S}_{\mathrm{tgt}} \to^{\varepsilon} \mathcal{S}'_{\mathrm{tgt}} \implies$
   $\exists \mathcal{S}'_{\mathrm{src}} \exists \alpha' \sqsupseteq \alpha.\ (\mathcal{C}_{\mathrm{src}}, N_{\mathrm{src}}) \vdash \mathcal{S}_{\mathrm{src}} \to^{\varepsilon} \mathcal{S}'_{\mathrm{src}} \wedge (\mathcal{S}'_{\mathrm{src}} \overset{\alpha'}{\sim} \mathcal{S}'_{\mathrm{tgt}})) \vee$

2. $(i_{\mathrm{src}} = \mathtt{ret}\ v_{\mathrm{src}} \wedge i_{\mathrm{tgt}} = \mathtt{ret}\ v_{\mathrm{tgt}} \wedge [\![v_{\mathrm{src}}]\!]_{s_{\mathrm{src}}} \overset{\alpha}{\simeq} [\![v_{\mathrm{tgt}}]\!]_{s_{\mathrm{tgt}}}) \vee$

3. $(i_{\mathrm{src}} = (x_{\mathrm{src}} = \mathtt{call}\ p_{\mathrm{src}}\ \vec{a}_{\mathrm{src}})\ \wedge\ i_{\mathrm{tgt}} = (x_{\mathrm{tgt}} = \mathtt{call}\ p_{\mathrm{tgt}}\ \vec{a}_{\mathrm{tgt}})$
   $\wedge\ [\![p_{\mathrm{src}}]\!]_{s_{\mathrm{src}}} \overset{\alpha}{\simeq} [\![p_{\mathrm{tgt}}]\!]_{s_{\mathrm{tgt}}}\ \wedge\ [\![\vec{a}_{\mathrm{src}}]\!]_{s_{\mathrm{src}}} \overset{\alpha}{\simeq} [\![\vec{a}_{\mathrm{tgt}}]\!]_{s_{\mathrm{tgt}}}\ \wedge$
   $(\forall \alpha' \sqsupseteq \alpha \forall \beta'_{\mathrm{src}} \forall \beta'_{\mathrm{tgt}} \forall h'_{\mathrm{src}} \forall h'_{\mathrm{tgt}} \forall v_{\mathrm{src}} \forall v_{\mathrm{tgt}}.$
   $(h'_{\mathrm{src}}, h'_{\mathrm{tgt}}) \in \mathcal{H}(\alpha', \beta'_{\mathrm{src}}, \beta'_{\mathrm{tgt}}) \wedge (v_{\mathrm{src}} \overset{\alpha'}{\simeq} v_{\mathrm{tgt}}) \wedge \implies$
   $(pc'_{\mathrm{src}}, s_{\mathrm{src}}[x_{\mathrm{src}} \mapsto v_{\mathrm{src}}], h'_{\mathrm{src}}) \overset{\alpha'}{\sim} (pc'_{\mathrm{tgt}}, s_{\mathrm{tgt}}[x_{\mathrm{tgt}} \mapsto v_{\mathrm{tgt}}], h'_{\mathrm{src}})))$.

where i_{src} and i_{tgt} are the instructions at the current pc of states $\mathcal{S}_{\mathrm{src}}$ and $\mathcal{S}_{\mathrm{tgt}}$ and pc' is the next line of pc.

This theorem says that if the source program does not get stuck, the target program should proceed and satisfies one of the three conditions: (i) for any target step, there is a corresponding source step, which results in the states again related by $\sim$ for some α' extending α; (ii) the programs are about to return the same values; or (iii) the programs are about to call the same function with the same arguments and, for any returned heaps and returned values related by α' extending α, the returned states are related by $\sim$ for α'.

We strongly believe that we can show that the program refinement result is derivable from this theorem. We have verified this simulation result for Simpleberry in Coq and and are currently proving the refinement result. See Section 7 for more details.

6 Evaluation

In this section, we evaluate our validator and confirm its extensibility and practicality. During the experiment, we used the machine with Linux 3.5 operating system equipped with a quad-core of Intel 2.8GHz box with 24GB of main memory. Since our validator is based on Vellvm with LLVM 3.0, it is also written in Coq 8.3pl1 and extracted code is compiled by OCaml 3.12.0.

File	Description (and Validated Optimizations)	#vali/#defd
InstructionCombining	Main of instruction combine pass + Common optimization functions (dead_elim, add_assoc)	2 / 32
InstCombineAddSub	Optimizations for add and sub instructions (add_signbit, add_zext_bool, add_onebit, add_shift, add_sub, add_sub2, add_mul_fold, add_const_not, add_select_zero, add_select_zero, sub_add, sub_onebit, sub_mone, sub_const_not, sub_zext_bool, sub_const_add, sub_remove, sub_remove2, sub_sdiv, sub_shl, sub_mul, sub_mul2)	22 / 46
InstCombineMulDivRem	Optimizations for mul, div, and rem instructions (mul_mone, mul_neg, mul_bool, mul_shl, div_sub_srem, div_sub_urem, div_zext, div_mone, rem_zext, rem_neg, rem_neg2)	11 / 39
InstCombineAndOrXor	Optimizations for bitwise binary instructions (and_zero, and_same, and_mone, or_xor)	4 / 55
InstCombineSelect	Optimizations for select instructions (select_trunc, select_add, select_const_gt)	3 / 13
InstCombineCompares	Optimizations for compare instructions (cmp_onebit, cmp_eq, cmp_ult)	3 / 31
InstCombine LoadStoreAlloca	Optimizations for memory access instructions (load_load, store_load)	2 / 24
InstCombinePHI	Optimizations for phi instructions (fold_phi_bin)	1 / 13
InstCombineCasts	Optimizations for cast instructions (trunc_onebit)	1 / 35
InstCombineShifts	Optimizations for shift instructions (shift_undef)	1 / 14
InstCombineCalls	Optimizations for call and invoke instructions	0 / 50
InstCombine SimplifyDemanded	Logic for simplifying instructions based on information for how they are used	0 / 42
InstCombineVectorOps	Optimizations for vector instructions	0 / 7
TOTAL		50 / 401

Table 1: Micro optimizations that we have validated.

Program	LOC[4]	Files	Comp	Gen	Vali	Hints	ParseErr
bc-1.06	13K	19	2.1s	2.3s	23.7s	860	0
tar-1.27	17K	67	5.8s	2.8s	31.4s	948	0
less-451	24K	35	3.9s	2.4s	22.9s	1119	0
make-4.0	28K	27	4.0s	7.5s	122.0s	2393	0
a2ps-4.14	52K	83	16.3s	11.4s	299.0s	2926	0
python-3.4.1	304K	221	79.7s	633.3s	12779.6s	33946	15

Table 2: Validation Time

[4] We counted only the number of lines of `.c` files without headers.

Simpleberry was extended to support 50 micro optimizations of the instruction combine pass at small cost. At the first stage of the development, we selected three representative micro optimizations, add_assoc, load_load, and fold_phi_bin, and built the infrastructure for Simpleberry to be able to validate the three optimizations. After building the infrastructure, we started to add more inference rules and update the hint converter in order to validate more optimizations. The extension from 3 to 50 optimizations took 3 person-weeks for validation of the 47 micro optimizations without verifying the inference rules, and 2 person-weeks for the correctness proof of 32 inference rules out of 63 rules. Among the unproven 31 inference rules, we simply did not prove 18 rules just because they involve tedious proofs about arithmetic operations, while they are intuitively obvious. The remaining 13 rules are unprovable under the current semantics of Vellvm due to an issue with undefined values. See Section 7 for more details about the issue.

As of now, Simpleberry covers 50 micro optimizations and they are categorized in Table 1. We strongly believe Simpleberry can be used to validate almost all micro optimizations of the instruction combine pass. Three of the categories have not been implemented for the following reasons.

- InstCombineCalls: it simplifies function calls to intrinsic functions by using preliminary knowledge of their semantics. However, the semantics of the intrinsic functions such as pow have not been axiomatized in Vellvm, so we left them as a future work.
- InstCombineSimplyDemanded: it simplifies instructions based on analysis results of demanded bits. In this case, we can validate the analysis results in Simpleberry by adopting inference rules about masking. However, we have not done it yet, because generating proper hints requires clear understanding of the analysis, which is not straightforward.
- InstCombineVectorOpts: it simplifies vector operations, but the vector type is not supported by Vellvm.

To show Simpleberry is practical, we applied it to our benchmarks and the results are presented in Table 2. Comp, Gen, Vali, and Hints denotes compile time, core hint generation time, validation time, and the number of generated hints, respectively. ParseErr denotes the number of source files that include unsupported syntax such as inline assembly and indirect branch. In our experiment, optimizations on 15 files were not validated for that reason. Except for the optimizations from the 15 files, all other 42000 generated hints were successfully validated in the experiment. The validation (Gen+Vali) took 5 to 168 times longer than the original compilation (Comp), but it can be parallelized as shown in Figure 1. Indeed, we performed the validations in parallel using 3 cores and added up the time consumed in the table above.

7 Discussion and Future Work

We discuss issues raised during our development of Simpleberry and future work.

Issues with Formal Semantics of LLVM In the course of the verification, we found several issues with both the formal semantics of Vellvm and the official semantics of LLVM. We discuss the issues and our opinion about them. Note that Vellvm provides two versions of formal semantics: deterministic one and non-deterministic one. The former is closer to the CompCert memory model and the latter is closer to the official semantics of LLVM. For our verification, we used the deterministic semantics because it is more natural than the other.

The deterministic semantics of Vellvm models the undefined value as a special value, called **undef**. This value is propagated through arithmetic operations. For example, we have $\mathbf{undef} + 1 = \mathbf{undef}$. The problem here is that we lose some information when such propagation occurs.

Due to this problem, we cannot prove the correctness of several optimizations in the instruction combine pass. As an example, consider the following optimization.

B1		
5: $x = a - 1$	$\mapsto$	$x = a - 1$
6: $y = a - x$	$\mapsto$	$y = a - x$
7: $z = y + 1$	$\mapsto$	$z = 1 + 1$

Intuitively, since $y = a - x = a - (a - 1) = 1$, we can replace y by 1. However, if a is **undef**, then $z =$ **undef** in the source and $z = 2$ in the target. This is problematic because, in the deterministic semantics, **undef** is interpreted as `false` in the branch while 2 is interpreted as `true`.

The official semantics of LLVM treats the undefined value as a set of all possible values. This makes the non-deterministic choice of undefined value as a lazy effect. For example, after executing $x =$ **undef**, if you read x twice, you may get different values. The purpose of this semantics is to allow to freely replace x by **undef** when you have $x =$ **undef**. However, this semantics leads to a somewhat weird optimization. Here is an example.

```
 1: unsigned char bar(unsigned char *i) {return *i;}
 2: unsigned char foo() {
 3:  unsigned char i;
 4:  bar(&i);
 5:  return i;
 6: }
 7: int main() {
 8:  unsigned int x = (unsigned int) foo();
 9:  printf(x < 2 ? "a" : "b");
10:  printf(x > 1 ? "a" : "b");
11:  return 0;
12: }
```

If you compile this program using clang with optimization level 2 (*i.e.,* -O2), the compiled program prints out `bb`. We tested this program with clang version 3.0 and 3.5.0. However, this optimization makes debugging hard because if you look at the code at line 9 and 10, you get contradiction because `x >= 2` and `x <= 1` should hold. Though not very clear, the standard C semantics seems to allow this because the program can go wrong at line 9 due to the uninitialization of the variable `i` at line 3. The reason why "bb" is printed is because LLVM propagates the effect of undefinedness raised for the variable i at line 3 to the variable x at lines 9 and 10 (*i.e.,* , replacing x by **undef**). Then both **undef** < 1 and **undef** > 0 are optimized to 0 to take the else branch.

Our plan is to model **undef** as an immediate effect of non-deterministic choice (not a special value nor a lazy effect). This simplifies the formal semantics and thus makes verification a lot easier. For example, we can have algebraic properties such as $x = a - 1 \wedge y = a - x \implies y = 1$, which is not the case in the presence of **undef** as discussed above. A down side is that we cannot allow propagating the **undef** command. We think this is not too bad because **undef** implicitly means that something went wrong and optimizations in an erroneous state might not be so useful.

There is another issue with LLVM optimization. The LLVM dead code eliminator removes a read-only function with even divergence effect. If you compile the example below using clang with optimization level 1 (*i.e.,* -O1), both in version 3.0 and 3.5.0, the compiled program prints out the "boom!" message.

```
1: void foo() { while (1) { } }
```

```
2: void bar() {
3:   foo();
4:   for (int i = 0; i < 100; ++i) { }
5: }
6: int main() {
7:   bar();
8:   printf("boom!\n");
9: }
```

Intuitively this code should not terminate without printing "boom!" because `foo()` diverges. The standard C semantics also does not allow this but for a little subtle reason. The C semantics basically allows to optimize away a benign infinite loop unless the loop condition is a constant. In the above example, since the loop condition at line 1 is a constant, it should not be removed.

Regardless of whether an infinite loop has a constant condition or not, optimizing away any infinite loop does not conform to the standard notion of program refinement and thus we cannot prove such a dead code optimization.

One possible direction is to provide a flag like -safe that prevents performing unsafe optimizations such as undef propagation and read-only function elimination unless we make sure that they are correct w.r.t. the formal semantics (*e.g.,* the read-only function always terminates). Another direction is to find a reasonable formal semantics that allows a benign infinite loop elimination.

Finally, LLVM has a notion of poison value, which is an undefined value with additional flag that it is poisonous. This value does not appear in the program syntax, but only in the dynamic semantics. We leave how to formalize poison values as a future work.

Current Status of Our Development We discuss the current status of development of Simpleberry.

First, we fully verified Theorem 1 in Coq 8.3 pl1 for Simpleberry assuming that the source and the target programs are well-formed and that functions with readonly tag do not modify the global and the parent's heap. The theorem in Coq corresponding to Theorem 1 requires the well-formedness of the source and the target program, and moreover, when the optimization relies on a function's readonlyness, the correctness of readonly tags.

Currently, we do not provide such properties and thus, technically speaking, our proof is incomplete. However, we can provide proofs for such properties by running a well-formedness checker and a readonlyness checker when necessary. Since well-formedness checking is syntax-direct and easy, we plan to develop such a checker in Coq and prove it correct. On the other hand, since readonlyness checking is quite complex, we plan to use the LLVM's readonlyness analyzer and validate its result by verified validation. We expect such additional checking would not take so long because well-formedness checking is very fast and, on the other hand, rather slow readonlyness checking is not invoked very often.

We also have memory fault bugs in our hint converter due to the foreign function interface (FFI) with C. In our hint converter, we use the LLVM bitcode parser for OCaml included in the official release, which invokes a parser written in C via FFI. We experience memory fault bugs when using the parser such as segmentation fault, out-of-memory or even divergence. We temporarily removed all such buggy behavior by inserting explicit GC calls, or strangely by enabling the debug flag in the parser. This is not a proper solution and we plan to properly debug and fixed the bugs.

Finally, there are several features of LLVM that Vellvm do not formalize. Among those, we only discuss features that appear during compilation of C source files. First, the switch instruction is not formalized and thus we remove it by running the lower-switch pass. Second, the inline assembly and the indirect branch instructions are not formalized and thus we could not validate any source files using such features. Third, Vellvm do not have concrete semantics for known functions like `pow` and thus we did not validate related optimizations such as optimizing `pow`$(x, 0)$ to 1. We can easily support it by axiomatizing the behavior of such functions. Fourth, Vellvm ignores flags to instructions that can raise poison values, such as nsw and inbounds flags.

Formalizing all these features is an interesting future work.

8 Related Work

Compiler verification has been a grand challenge [2] and there has been a great amount of work that made such a goal feasible.

Verified Compilation The most important step towards compiler verification is Leroy's seminal work on the CompCert compiler [4, 8]. It is a fully verified C compiler written in Coq. Despite of various efforts, there has been no bug found for verified parts of the compiler [11, 3]. Our LLVMBERRY project is strongly motivated by this result.

In particular, the CompCert memory model has a great influence on other compiler verification work. For example, Vellvm's memory model is also based on the CompCert model. We also used several lemmas from the CompCert project for our verification of SIMPLEBERRY.

LLVM Formalization and Verification The Vellvm framework [12] provides a formalization of LLVM IR and our work is directly built up on this framework. Vellvm has been also used to verify the mem2reg pass of the LLVM compiler [13]. However, for this verification, the mem2reg pass is rewritten and verified in Coq. The reason for re-implementation is that the original mem2reg pass temporarily breaks the well-formedness of LLVM IR.

We believe, however, that it is possible to develop a verified validator for the original mem2reg pass because, in verified validation, we do not need to show the preservation of well-formedness. Instead, we just need to check the well-formedness using a well-formedness checker whenever the well-formedness is recovered. In other words, it is fine for well-formedness to be broken temporarily as long as we do not rely on the property at that point.

We believe this because indeed SIMPLEBERRY does not rely on the full SSA property, but only on the well-typedness.

Verified Validation Tristan and Leroy [8, 9, 10] first applied the verified validation approach to several compiler optimizations such as instruction scheduling, lazy code motion, and software pipelining. In this work, they have shown that verified validation can be much more effective than verified compilation for some optimizations.

Barthe et al. [1] presented a formally verified SSA-based, middle-end compiler on CompCert. In this work, they applied the verified validation to a complex global value numbering (GVN) optimization. This is another example where verified validation is more effective than verified compilation.

Translation Validation The translation validation approach is to develop a general-purpose algorithm that checks the validity of compilation results, which motivated the verified validation approach. Necula [6] developed an algorithm using symbolic evaluation and Tristan et al. [7] developed a translation validator based on a value-graph algorithm and applied it to several optimization passes in LLVM.

A downside of this approach is that the algorithm is not complete. For example, the validator developed by Tristan et al. [7] has not been applied to the instruction combine pass. More importantly, since the validator is not verified, we cannot trust the validation result with the highest confidence.

References

[1] G. Barthe, D. Demange, and D. Pichardie. A formally verified SSA-based middle-end: Static single assignment meets Compcert. In *ESOP*, 2012.

[2] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, Jan. 2003.

[3] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *PLDI*, 2014.

[4] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[5] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41(1):1–31, July 2008.

[6] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.

[7] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI*, 2011.

[8] J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *POPL*, 2008.

[9] J.-B. Tristan and X. Leroy. Verified validation of lazy code motion. In *PLDI*, 2009.

[10] J.-B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *POPL*, 2010.

[11] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[12] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. *SIGPLAN Not.*, 47(1):427–440, Jan. 2012.

[13] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *PLDI*, 2013.