Sound Translation Validation for LLVM

Thesis submitted by

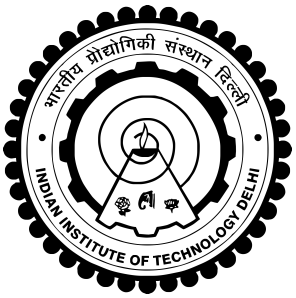
Aditya Senthilnathan
2017CS50403

under the guidance of

Prof. Sorav Bansal, Indian Institute of Technology Delhi

in partial fulfilment of the requirements
for the award of the degree of

Bachelor and Master of Technology



Department Of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY DELHI

March 2022

THESIS CERTIFICATE

This is to certify that the thesis titled **Sound Translation Validation for LLVM**, submitted by **Aditya Senthilnathan (2017CS50403)**, to the Indian Institute of Technology, Delhi, for the award of the degree of **Bachelor and Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Sorav Bansal
Associate Professor
Dept. of CSE
IIT-Delhi, 110 016

Place: New Delhi

Date: June 13, 2022

ACKNOWLEDGEMENTS

I would like to use this page to express my gratitude to the people who have supported me, directly or indirectly, during my undergraduate studies.

First and foremost, I would like to thank my advisor, Prof. Sorav Bansal, who introduced me to research in programming languages and compilers. As a mentor, he has significantly shaped my perspectives on research and work ethic, which I will likely hold on to for the rest of my career. I've learned a tremendous amount from him, such as how to approach and think about big problems, build complicated systems one component at a time, circumnavigate dead-ends and more that are difficult to put into words. Working with him and learning from him these past two years has been a wonderful and enriching experience. I would like to express gratitude to Prof. Subodh Sharma, Prof. Abhilash Jindal, and Prof. Uday Reddy Bondhugula for evaluating and giving me constructive feedback on my research.

I'm grateful to my parents and brother for their steadfast support during all the ups and downs of my undergraduate studies. Without them, none of this would have been possible.

I would like to thank Abhishek Rose, Indrajit Banerjee, and Ritesh Saha, with whom I had stimulating and delightful discussions on various things. In particular, I would like to thank Abhishek for being extremely helpful and always making time to discuss things with me whenever I was facing technical challenges in my research.

Finally, I am grateful to all my friends, who made my time in Delhi lively. I am thankful for the wonderful memories we've made and shall cherish them.

ABSTRACT

KEYWORDS: Translation Validation, Undefined Behavior, Compilers, Automatic Verification, Intermediate Representation (IR).

We have designed and implemented `superopt-refines` — an automatic sound translation validation prototype for the LLVM intermediate representation (LLVM IR). As a translation validator, `superopt-refines` certifies an execution of an LLVM optimizer by verifying refinement of the source (optimizer input) and target (optimizer output) programs. `superopt-refines` is made automatic through the use of an SMT solver, is designed to be sound, and requires no changes to LLVM. In evaluating `superopt-refines` against `alive-tv` [LLH+21] (a bounded automatic translation validator for LLVM), we find that due to `superopt-refines`'s sound modeling of LLVM IR semantics, it is able to disprove the validity of 18 transformations that are admitted as valid by `alive-tv`. Through `superopt-refines`, we demonstrate the benefits and problems encountered in sound modeling of LLVM IR semantics (particularly undefined behavior semantics).

Contents

List of Tables

List of Figures

ABBREVIATIONS

UB	Undefined Behavior
SR	superopt-refines
AT	alive-tv
pc	Poison Condition Predicate

NOTATION

$\wedge$	Logical And Operator
$\vee$	Logical Or Operator
$\oplus$	Exclusive Or (XOR) Operator
$\sqsupseteq_u$	Refinement Operator for `undef` values
$\sqsupseteq$	Refinement Operator
wp	Weakest Precondition Function

Chapter 1

Introduction

The LLVM project is a popular collection of compiler and toolchain technologies. One of the subprojects of the LLVM project is the LLVM Core libraries, which provides an optimizer that is independent of both the source and target programming languages. Optimization independent of the source and target languages is achieved by building the LLVM Core libraries around a well-specified code representation - the LLVM Intermediate Language (LLVM IR). Simply put, the source language provides a front-end that compiles to LLVM IR; programs in LLVM IR are optimized by LLVM Core libraries and then high-quality code can be generated for a variety of target architectures from the optimized LLVM IR code.

Despite the popularity and widespread usage of the LLVM project, the LLVM IR has certain issues concerning its specification, implementation, and usage of undefined behavior such as inconsistencies in specified semantics, incorrect optimizations, points of disagreement between specification and implementation, ambiguities in the specification, and fundamental flaws in the design of the IR, as established by [LKS+17, LMNR15, LLH+21]. Resolving these issues related to undefined behavior is important as LLVM optimizers take advantage of undefined behavior frequently. Furthermore, undefined behavior is important for languages to communicate invariants about programs to the optimizers in order to enable further optimization.

One solution to address this problem would be to formalize LLVM IR semantics from their high-level specification [Dev14], and certify individual executions of the optimizer as per the formalized semantics — translation validation. Previous works [LMNR15, LLH+21] have built fully automatic translation validation tools for LLVM and demonstrated the utility of such tools by applying them to verify and discover incorrect optimizations in LLVM. However, these tools are unsound as they have made certain under-approximating assumptions that allow them to admit invalid optimizations as demonstrated later in Section 4.3.

We propose an automatic *sound* translation validation prototype for LLVM — `superopt-refines`. It verifies optimizations by checking pairs of functions in LLVM IR — the source and target of the optimization — for refinement. The source and target function satisfy a refinement relation if and only if for every possible input state, the target function displays only a subset of the behaviors of the source function [LMNR15, LLH+21]. Being a sound translation validator, `superopt-refines` has been able to detect invalid optimizations that have been admitted as valid by tools such as Alive2 [LLH+21].

Being a prototype, `superopt-refines` is limited in the language features of LLVM IR that it supports. The primary intention of `superopt-refines` is to demonstrate how undefined behavior can be modeled in a sound manner, and the problems and benefits of such modeling.

Chapter 2

Background

2.1 LLVM's Undefined Behavior

LLVM has two categories of undefined behavior (UB) [LKS+17]:

1. **Immediate UB**: This is the strongest form of UB and is triggered for serious errors such as divide-by-zero, dereferencing an invalid pointer, etc. If immediate UB is encountered in a program, the compiler is allowed to do anything including "to make demons fly out of your nose". Realistically speaking, it results in effects like exceptions, processor traps and RAM corruption.
2. **Deferred UB**: This form of UB describes operations that produce unpredictable values but are otherwise safe to execute. Deferred UB is necessary to support speculative execution such as hoisting potentially undefined operations out of loops. It comes in two forms in LLVM:
   (a) `poison`
   (b) `undef`

2.1.1 Immediate UB

Why does LLVM need immediate UB?

LLVM is an intermediate representation (IR) language and as such it needs to be able to express semantics from various languages. Immediate UB gives language designers the freedom to choose whether to define the behavior of operations that are logically not well defined such as divide-by-zero. For example, languages like Java and Python generate exceptions for divide-by-zero whereas languages like C/C++ define it to be immediate UB for performance benefits.

Furthermore, immediate UB makes more compiler optimizations[1] possible in LLVM . One such optimization is demonstrated below[2].

[1] The terms "transformation" and "optimization" are used interchangeably.

[2] Transformations are sometimes demonstrated in C++ for brevity. The same principles would apply if the code is in LLVM.

Listing 2.1: Source

```
int n = ...;
for(int i=0; i<n+1; i++){
    ...
}
```

Listing 2.2: Target

```
int n = ...;
for(int i=0; i<=n; i++){
    ...
}
```

Note: The original program is called the source program and the optimized program is called the target program in a transformation/optimization.

In the source program, it is possible for `n+1` to have signed overflow. If we assumed signed overflow to have wrap-around semantics (Ex. `INT_MAX+1` would wrap-around to 0), then the above transformation is illegal because the source and target program have different behaviors — if `n = INT_MAX`, in the source program, the loop will never execute whereas in the target program, the loop will never terminate.

However, if we assumed that signed overflow is immediate UB, this transformation is legal. With these semantics, if `n = INT_MAX`, `n+1` in the source program would trigger immediate UB, and therefore the compiler is allowed to replace `i < n+1` with `i <= n`. Thus, immediate UB allows this transformation to be legal.

While immediate UB enables some transformations, it also disables some transformations as shown below.

Listing 2.3: Source

```
int x = ...; int y;
bool cond = ...;
while(cond){
    y = x+1;
    ...
}
```

Listing 2.4: Target

```
int x = ...; int y;
bool cond = ...;
y = x+1;
while(cond){
    ...
}
```

This hoisting transformation is illegal because if `x = INT_MAX` and `cond = false`, the behaviors of the source and target programs diverge - `x+1` triggers immediate UB in the target program but the source program doesn't exhibit immediate UB because the loop doesn't execute.

2.1.2 Poison Values

What are poison values?

A `poison` value is a special value that represents a violation of an assumption. An operation on a `poison` value either results in a `poison` value or triggers immediate UB. For example, in the following LLVM code,

```
%a = sub nuw i32 0, 1;      // poison
%b = add i32 %a, 1;         // poison
%c = and i32 %b, 1;         // poison
%d = store i32 0, i32* %c;  // Immediate UB
```

The register `%a` yields `poison` value. This is because, the `nuw` flag in the subtract (`sub`) operation encodes the assumption that unsigned overflow should not occur in this operation. However, subtracting 1 from 0 triggers unsigned overflow and thus the result of the `sub` operation is `poison`. Registers `%b` and `%c` are `poison` as per LLVM semantics [Dev14] because one of the operands in those instructions are `poison` values. Storing to an address that is a `poison` value triggers immediate UB as per LLVM semantics. As a result, the instruction assigned to register `%d` triggers immediate UB since it is trying to store 0 to address `%c` which is `poison`.

LLVM's definition of `poison` semantics can be found here

Why does LLVM need poison values?

`poison` values are a weaker and less non-deterministic form of UB than immediate UB and they help enable some transformations that would otherwise be disabled with immediate UB. For example, we can see that the transformation Listing 2.3 → 2.4 is illegal when signed overflow is immediate UB. However, transformations such as these are common and profitable and we'd like for such transformations to be legal. We can achieve this through poison value semantics. If the transformation 2.3 → 2.4 were to be such that signed overflow emits poison value instead of immediate UB, then the transformation becomes legal (shown below).

Listing 2.5: Source

```
int x = ...; int y;
bool cond = ...;
while(cond){
    y = add nsw i32 x, 1;
    ...
}
```

Listing 2.6: Target

```
int x = ...; int y;
bool cond = ...;
y = add nsw i32 x, 1;
while(cond){
    ...
}
```

When `x = INT_MAX` and `cond = false`, `y` yields `poison` and this `poison` value isn't used anywhere in the target program. As a result, immediate UB is not triggered in the target program (unlike in transformation 2.3 → 2.4).

2.1.3 Undefined Values

An undefined value in LLVM (also called `undef`) indicates that the user of the value may receive an unspecified bit-pattern. `undef` values can be used anywhere a constant is expected. For example, in the following LLVM code,

```
%a = add i8 undef, 0;
%b = mul i8 %a, 2;
```

The `undef` value here represents an unspecified bit-pattern of bitwidth 8 and so, it can yield any value in `{0,1,2...255}`. Since `undef` can yield any value in `{0,1,2...255}`, `%a` can also yield any value in `{0,1,2...255}`. Since `%a` is of type `i8`, and can yield any value permissible within the type `i8` (bitvector of bitwidth 8), we call `%a` and such values "fully `undef`". As for `%b`, since `%a` is multiplied by 2, `%b` could yield any value in `{0,2...254}` (the set of even values). Unlike `%a`, the register `%b` is limited in what values it can yield and so, we call such values "partially `undef`".

Every time an `undef` value is observed, it can yield a different value from its set of values. For example, the following pseudo-code could print 10 different numbers from the set $\{0,1,2\ldots2^{32}-1\}$.

```
int a = undef;
for (int i=0; i<10; i++){
    print(a);
}
```

LLVM's definition of `undef` semantics can be found here

Why do we need undef values?

The LLVM IR caters to multiple languages and as such, has to be able to express different language semantics. However, the LLVM IR is incapable of expressing certain language semantics with just immediate UB and `poison`. Consider the following LLVM pseudocode example:

```
int x;
y = x*0;
print(y);
```

Some languages expect random/arbitrary values for uninitialized variables. Such languages would expect the above code to print 0 and not print an error value or trigger immediate UB. However, we can't express these semantics in LLVM IR with just immediate UB or poison - if the uninitialized variable x is made a poison, then y also yields a poison value and trying to print a poison value would result in immediate UB. To be able to print 0 with the above code example and express such language semantics in LLVM IR, we need undefined values which are a weaker and less non-deterministic form of UB than `poison`. In the above code example, if x is made `undef`, y would yield 0 and the program would print 0 as expected.

2.1.4 `freeze` Instruction

The `freeze` instruction was introduced due to [LKS+17] and the efforts of its authors. As described in [LKS+17], this instruction is used to stop the propagation of `undef` and `poison` values by converting them to well-defined values. The instruction takes a single argument and if that argument is `poison` or `undef`, it returns an arbitrary but fixed and well-defined value of that argument's type. If the argument is instead well-defined, then the instruction is a no-op and the input argument is returned. Ex. In `%a = freeze i8 undef`, register `%a` and all its subsequent uses might choose to yield the fixed and well-defined value `5` which is a value of type `i8`. All uses of the same `freeze` instruction will yield the same value (unlike `undef` values), while different `freeze` instructions may yield different values.

2.1.5 Relative Degree of Non-Determinism of UB in LLVM

The different forms of UB in LLVM have varying degrees of non-determinism and we can impose an ordering upon them based on the relative degree of their non-determinism. The relative degree of non-determinism in LLVM is as shown below,

`Immediate UB > poison > undef > well-defined values`

Immediate UB exhibits the most non-deterministic behavior since the compiler can do anything on encountering immediate UB. Well-defined values are the least non-deterministic since the behavior of these values is clearly defined and deterministic.

2.1.6 Correctness of Transformations

A compiler transformation is correct if and only if the target (optimized) code only exhibits behaviors that are also exhibited by the source (original) code. Transformations can be verified by establishing a refinement relation between the source and target code.

A refinement relation between functions is satisfied when, for every possible input state, the target function displays only a subset of the behaviors of the source function [LMNR15, LLH+21]. Informally speaking, refinement allows a transformation to remove non-determinism from the optimized program, but not add it. In the absence of undefined behaviors, refinement degenerates to simple equivalence.

Example 1

A compiler is allowed to convert a value to a less non-deterministic value in a transformation. For example, a `poison` value could be converted an `undef` value because `undef` is less non-deterministic than `poison`. Similarly, a `poison` or `undef` value could be converted to a well-defined value. However, a value cannot be converted to a value that is more non-deterministic. For example, an `undef` value or well-defined value cannot be converted to a `poison` value. The following are some legal ($\rightarrow$) and illegal ($\not\rightarrow$) transformations,

$$
\begin{aligned}
a = poison &\rightarrow a = undef \\
b = poison &\rightarrow b = 1 \\
c = undef &\rightarrow c = 1 \\
d = \frac{x}{0} &\rightarrow d = 1 \\
e = undef &\rightarrow e = undef \times 2 \\
f = undef &\not\rightarrow f = poison \\
g = 1 &\not\rightarrow g = undef
\end{aligned}
$$

Example 2

Listing 2.7: Source

```
define i8 @fn(i8 %x){
    %a = mul i8 %x, 2;
    ret i8 %a;
}
```

Listing 2.8: Target

```
define i8 @fn(i8 %x){
    %a = add i8 %x, %x;
    ret i8 %a;
}
```

The above transformation is a simple optimization that utilizes the knowledge that `x+x == 2*x`. However, this transformation is invalid. This is because the transformation introduces more non-determinism into the program — if `x = undef`, then `%a` in the source program can only yield a value from the set `{0,2...254}` (the set of even values) but `%a` in the target program can yield any value from the set `{0,1,2...255}` (all values of `i8`). Since the return value of the target program can exhibit values that the return value of the source program cannot, the target program exhibits behaviors not exhibited by the source program. Thus, the source and target programs do not satisfy a refinement relation and this transformation is invalid.

Chapter 3

superopt-refines

In this chapter, we describe our proposed automatic sound translation validation prototype, `superopt-refines`, for verification of source-to-source transformations in LLVM IR.

Currently, `superopt-refines` can only verify transformations on functions that do not contain branches, loops, heap-manipulating instructions, and freeze instructions and supports only integer types.

3.1 Overview

When given a source and target function in LLVM IR, `superopt-refines` verifies the correctness of the transformation from source function to target function, and declares whether the transformation is valid or not.

To verify the correctness of a transformation from source to target function, `superopt-refines` first constructs individual transfer functions for the two functions, each instrumented with predicates encoding information related to LLVM's undefined behavior semantics. Following this, `superopt-refines` uses the transfer functions to construct a formula in SMT that can, if proven, verify that the transformation is valid. This SMT formula is then passed to an SMT solver which attempts to prove it. Based on whether the SMT solver could prove the SMT formula, `superopt-refines` declares whether the transformation is valid or not.

This process is described in more detail, in Sections 3.2 – 3.4

3.2 Transfer Function

Given source and target functions in LLVM IR, `superopt-refines` constructs individual transfer functions for the two functions, each instrumented with predicates. This section discusses the predicates instrumented in the transfer function for a function in LLVM IR by `superopt-refines`.

3.2.1 Assumes Predicates

An assumes predicate specifies the conditions under which an instruction is well-defined i.e. it doesn't trigger immediate UB [DB17b]. These predicates are instrumented in the

transfer function for each instruction that could cause immediate UB.

The assumes predicates that `superopt-refines` encodes for various instructions have been adopted from the definedness constrains of the Alive tool [LMNR15].

Example 1:

LLVM Program:

```
C0: define i8 @fn(i8 %x){
C1:     %a = udiv i8 1, %x;
C2:     %b = sub i8 %a, 1;
C3:     %c = udiv i8 1, %b;
C4:     ret i8 %c;
C5: }
```

Corresponding Transfer Function:

```
a := udiv(1, x)
b := sub(a, 1)
c := udiv(1, b)
ret_val := c
```

Corresponding Assumes Predicates:

```
{x ≠ 0, b ≠ 0}
```

Division by zero triggers immediate undefined behavior in LLVM and hence for the program in Example 1, we have the assumes predicates `{x ≠ 0, b ≠ 0}` which encode the conditions under which the program is well-defined.

3.2.2 Poison Condition Predicates

A poison condition predicate specifies the conditions under which an instruction yields `poison` value. These predicates are instrumented in the transfer function for each instruction that could yield `poison`.

The poison condition predicates that `superopt-refines` encodes for various instructions have been adopted from the constraints for poison-free execution of the Alive tool [LMNR15].

Example 2:

LLVM Program:

```
C0: define i8 @fn(i8 %x, i8 %y){
```

```
C1:     %a = add i8 nuw %x, %y;
C2:     %b = udiv i8 1, %a;
C3:     ret i8 %b;
C4: }
```

Corresponding Transfer Function:

```
a := add(x, y)
a.pc := x.pc ∨ y.pc ∨ (zext(x,1) + zext(y,1) ≠ zext(x + y, 1))
b := udiv(1, a)
b.pc := a.pc
ret_val := b
```

Corresponding Assumes Predicates:

```
{a ≠ 0}
```

`a.pc` represents the poison condition predicate for the register `%a` and likewise for other registers in the program. The poison condition predicates have been instrumented in the transfer function alongside the encoding of the actual instructions in the program.

Register `%a` can yield poison value if either register `%x` or `%y` is a poison value which is why `%a`'s poison condition predicate has the term `x.pc ∨ y.pc` (`x.pc` and `y.pc` are encoded as uninterpreted constants in the final SMT formula). Register `%a` can also yield poison value if registers `%x` and `%y` cause unsigned overflow on addition and hence, `%a`'s poison condition predicate also contains the term `(zext(x,1) + zext(y,1) ≠ zext(x + y, 1))`.

3.3 Encoding `undef` semantics in SMT

Due to the existence of fully and partially undefined values in LLVM IR, a register needs to be modeled as a set of values that it can potentially yield instead of as a single value. This section discusses `superopt-refines`'s scheme for encoding LLVM IR's `undef` semantics in SMT.

The set of values that a register can yield as a fully or partially `undef` value is modeled in `superopt-refines` by encoding the corresponding set membership function in SMT using an uninterpreted function or lambda function of type `(register type → boolean)`. For example, we could represent a register `i8 x` that is partially `undef` and can only be an even value as `x.set: i8 → bool` where,

```
x.set := λ (c: i8). c mod 2 == 0
```

3.3.1 Function Arguments

A function argument can be a fully `undef` value or any partially `undef` value i.e. it can yield any set of values and the encoding needs to appropriately reflect this. For this reason, for a function argument of type `T`, we create an uninterpreted function of type `(T → bool)` to represent the set membership function for this argument's set of values, and we do not impose any constraints on this function. Since the uninterpreted function can represent any set of values permissible in type `T`, it can represent any partially or fully `undef` value for the function argument.

3.3.2 Instructions

We encode the set membership functions for values of instructions such as `add, sub`, etc., as lambda functions. For example take the following arbitrary non-heap-manipulating binary operation of type `T` (can be in LLVM IR or transfer function),

```
op T %a, %b
(or)
op(a, b)
```

Assume that the set membership functions for `%a` and `%b` are already defined as `a.set` and `b.set` respectively. We encode the set membership function for the result of this operation as a lambda function of type `(T → bool)`, as follows,

$$\lambda\ \texttt{(m: T)}.\ \exists\ \texttt{x, y.}\ \texttt{a.set(x)} \wedge \texttt{b.set(y)} \wedge \texttt{(op(x, y) == m)}$$

3.3.3 Constants and Well-defined Values

For the sake of uniformity, we consider constants and well-defined values as singleton sets and accordingly encode set membership functions for them. For example, for a constant value `v` of type `T`, we encode the set membership function as λ `(m: T). m == v`.

3.4 Verification Condition in SMT

This section discusses the SMT formula to be verified and how `superopt-refines` constructs it, given the instrumented transfer functions of the source and target functions.

Before we introduce the verification condition, we first introduce some required operators.

3.4.1 Operator $\sqsupseteq$

Let the operator $\sqsupseteq_u$ be defined as follows:

$$\frac{\forall x.\ b.set(x) \implies a.set(x)}{a \sqsupseteq_u b}$$

$a.set$ represents the set membership function for the set of values that register a can yield, and the same applies for $b.set$ and register b.

In the context of `superopt-refines`, we define the operator $\sqsupseteq$ (refinement operator) to be as follows:

$$\frac{(b.pc \implies a.pc) \wedge (a.pc \vee a \sqsupseteq_u b)}{a \sqsupseteq b}$$

The operands of the $\sqsupseteq$ and $\sqsupseteq_u$ operators are registers in the source and target functions. $a.pc$ represents the poison condition predicate for the instruction corresponding to register a, and similarly for $b.pc$ and register b.

In the context of `superopt-refines`, a register b refines register a, or $a \sqsupseteq b$ if and only if (1) register b yields poison only if register a also yields poison, (2) If register a does not yield poison, the set of values register b can yield must be a subset of the values that register a can yield. Invariants involving the $\sqsupseteq$ operator are informally called refinement invariants.

3.4.2 Verification Condition

Of the input source function and input target function, let the function arguments for the source function be $src.arg_1$, $src.arg_2$, $\ldots src.arg_n$ and for the target function be $tgt.arg_1$, $tgt.arg_2$, $\ldots tgt.arg_n$. Let the return value for the source function be $src.ret\_val$ and for the target function be $tgt.ret\_val$. Let the set of assumes predicates for the source function be $src.assumes\_preds$ and for the target function be $tgt.assumes\_preds$. Let the transfer functions for the source and target function be tf_{src} and tf_{tgt} respectively. Let the weakest precondition function be denoted by wp.

Let $src.assumes\_conjunct$ and $tgt.assumes\_conjunct$ be defined as follows,

$$src.assumes\_conjuct = \bigwedge_{pred \in src.assumes\_preds} wp(tf_{src}, pred)$$

$$tgt.assumes\_conjuct = \bigwedge_{pred \in tgt.assumes\_preds} wp(tf_{tgt}, pred)$$

Assuming that the source and target functions do not contain any branches, loops, heap-manipulating instructions, or freeze instructions, and contains only integer types, the verification condition for proving that the target function refines the source function consists of the following two predicates,

1. $(src.assumes\_conjunct \implies tgt.assumes\_conjunct)$
2. $(src.assumes\_conjunct \wedge tgt.assumes\_conjunct \wedge (\bigwedge_{i=1}^{n} src.arg_i \sqsupseteq tgt.arg_i)) \implies wp(tf_{src},\ src.ret\_val) \sqsupseteq wp(tf_{tgt},\ tgt.ret\_val)$

If these two predicates are proven, then the transformation from the input source function to the input target function in LLVM IR is valid.

3.5 Example Transformations

3.5.1 Transformation 1

Source Function:

```
define i8 @fn(i8 %x){
    %a = add i8 %x, %x;
    ret i8 %a;
}
```

Source Transfer Function:

```
a := add(x, x)
a.pc := x.pc
ret_val := a
```

Source Assumes Predicates: {}

Target Function:

```
define i8 @fn(i8 %x){
    %a = mul i8 %x, 2;
    ret i8 %a;
}
```

Target Transfer Function:

```
a := mul(x, 2)
a.pc := x.pc
ret_val := a
```

Target Assumes Predicates: {}

Verification Condition:

1. $(src.assumes\_conjunct \implies tgt.assumes\_conjunct)$
   $\Leftrightarrow$ True $\implies$ True
   $\Leftrightarrow$ True
2. $(src.x \sqsupseteq tgt.x) \implies (wp(tf_{src},\ src.ret\_val) \sqsupseteq wp(tf_{tgt},\ tgt.ret\_val))$
   $\Leftrightarrow (src.x \sqsupseteq tgt.x) \implies (add(src.x, src.x) \sqsupseteq mul(tgt.x, 2))$
   $\Leftrightarrow ((tgt.x.pc \implies src.x.pc) \wedge (src.x.pc \vee src.x \sqsupseteq_u tgt.x)) \implies$
   $((tgt.x.pc \implies src.x.pc) \wedge (src.x.pc \vee add(src.x, src.x) \sqsupseteq_u mul(tgt.x, 2)))$

   $\vdots$

   $\Leftrightarrow$ True

Thus, the target function refines the source function and this transformation is valid.

For the term $add(src.x, src.x) \sqsupseteq_u mul(tgt.x, 2))$, `superopt-refines` would define the following set membership functions,

$$\begin{aligned} src.a.set &\mapsto \lambda m.\ \exists\ p, q.\ src.x.set(p) \wedge src.x.set(q) \wedge (p + q == m) \\ const2.set &\mapsto \lambda m.\ m == 2 \\ tgt.a.set &\mapsto \lambda m.\ \exists\ p, q.\ tgt.x.set(p) \wedge const2.set(q) \wedge (p \times q == m) \end{aligned}$$

Using these set membership functions, `superopt-refines` would expand the term $add(src.x, src.x) \sqsupseteq_u mul(tgt.x, 2))$ in the verification condition in the following manner,

$$\begin{aligned} add(src.x, src.x) \sqsupseteq_u mul(tgt.x, 2)) \\ \Leftrightarrow \forall x.\ tgt.a.set(x) \implies src.a.set(x) \end{aligned}$$

3.5.2 Transformation 2

Source Function:

```
define i8 @fn(i8 %x){
    %a = udiv i8 %x, 0;
    ret i8 %a;
}
```

Source Transfer Function:

```
a := udiv(x, 0)
a.pc := x.pc
ret_val := a
```

Source Assumes Predicates: $\{0 \neq 0\}$

Target Function:

```
define i8 @fn(i8 %x){
    ret i8 1;
}
```

Target Transfer Function:

```
ret_val := 1
```

Target Assumes Predicates: {}

Verification Condition:

1. $(src.assumes\_conjunct \implies tgt.assumes\_conjunct)$
   $\Leftrightarrow (0 \neq 0) \implies True$
   $\Leftrightarrow False \implies True$
   $\Leftrightarrow True$
2. $(src.assumes\_conjunct \wedge tgt.assumes\_conjunct \wedge \ldots) \implies \ldots$
   $\Leftrightarrow (0 \neq 0) \wedge True \wedge \ldots \implies \ldots$
   $\Leftrightarrow False \implies \ldots$
   $\Leftrightarrow True$

Thus, the target function refines the source function and this transformation is valid.

3.5.3 Transformation 3

Source Function:

```
define i8 @fn(i8 %x){
    %a = add i8 %x, %x;
    ret i8 %a;
}
```

Source Transfer Function:

```
a := add(x, x)
a.pc := x.pc
ret_val := a
```

Source Assumes Predicates: {}

Target Function:

```
define i8 @fn(i8 %x){
    %a = add nuw i8 %x, %x;
    ret i8 %a;
}
```

Target Transfer Function:

```
a := add(x, x)
a.pc := x.pc ∨ (zext(x,1) + zext(x,1) ≠ zext(x + x, 1))
ret_val := a
```

Target Assumes Predicates: {}

Verification Condition:

1. $(src.assumes\_conjunct \implies tgt.assumes\_conjunct)$
   $\Leftrightarrow$ True $\implies$ True
   $\Leftrightarrow$ True
2. $(src.x \sqsupseteq tgt.x) \implies (wp(tf_{src},\ src.ret\_val) \sqsupseteq wp(tf_{tgt},\ tgt.ret\_val))$
   $\Leftrightarrow (src.x \sqsupseteq tgt.x) \implies (add(src.x, src.x) \sqsupseteq add(tgt.x, tgt.x))$
   $\Leftrightarrow ((tgt.x.pc \implies src.x.pc) \wedge (src.x.pc \vee src.x \sqsupseteq_u tgt.x)) \implies$
   $((tgt.a.pc \implies src.a.pc) \wedge (src.a.pc \vee add(src.x, src.x) \sqsupseteq_u add(tgt.x, tgt.x)))$
   $\Leftrightarrow ((tgt.x.pc \implies src.x.pc) \wedge (src.x.pc \vee src.x \sqsupseteq_u tgt.x)) \implies$
   $((tgt.x.pc \vee (zext(tgt.x, 1) + zext(tgt.x, 1) \neq zext(tgt.x + tgt.x, 1))) \implies src.x.pc) \wedge \ldots)$

   $\vdots$

   $\Leftrightarrow$ False

The terms colored in red are not provable with $(src.x \sqsupseteq tgt.x)$. Hence the target function does not refine the source function and this transformation is invalid. A counterexample for this transformation is $\{src.x.set \mapsto \lambda m.\ m == 255,\ tgt.x.set \mapsto \lambda m.\ m == 255,\ src.x.pc \mapsto False,\ tgt.x.pc \mapsto False\}$ which would make the target function return `poison` value but not the source function.

3.5.4 Transformation 4

Source Function:

```
define i8 @fn(i8 %x){
    %a = add i8 %x, %x;
    ret i8 %a;
}
```

Source Transfer Function:

```
a := add(x, x)
a.pc := x.pc
ret_val := a
```

Source Assumes Predicates: {}

Target Function:

```
define i8 @fn(i8 %x){
    %a = mul i8 %x, 3;
    %b = sub i8 %a, %x;
    ret i8 %b;
}
```

Target Transfer Function:

```
a := mul(x, 3)
a.pc := x.pc
b := sub(a, x)
b.pc := a.pc ∨ x.pc
ret_val := b
```

Target Assumes Predicates: {}

Verification Condition:

1. $(src.assumes\_conjunct \implies tgt.assumes\_conjunct)$
   $\Leftrightarrow$ True $\implies$ True
   $\Leftrightarrow$ True
2. $(src.x \sqsupseteq tgt.x) \implies (wp(tf_{src},\ src.ret\_val) \sqsupseteq wp(tf_{tgt},\ tgt.ret\_val))$
   $\Leftrightarrow (src.x \sqsupseteq tgt.x) \implies (add(src.x, src.x) \sqsupseteq sub(mul(tgt.x, 3), tgt.x))$
   $\Leftrightarrow ((tgt.x.pc \implies src.x.pc) \wedge (src.x.pc \vee src.x \sqsupseteq_u tgt.x)) \implies$
   $((tgt.b.pc \implies src.a.pc) \wedge (src.a.pc \vee add(src.x, src.x) \sqsupseteq_u sub(mul(tgt.x, 3), tgt.x)))$
   $\Leftrightarrow ((tgt.x.pc \implies src.x.pc) \wedge (src.x.pc \vee src.x \sqsupseteq_u tgt.x)) \implies$

$$((tgt.x.pc \implies src.x.pc) \wedge (src.x.pc \vee add(src.x, src.x) \sqsupseteq_u sub(mul(tgt.x, 3), tgt.x)))$$

$$\vdots$$

$\Leftrightarrow$ False

The terms colored in red are not provable with $(src.x \sqsupseteq tgt.x)$. Hence, the target function does not refine the source function and this transformation is invalid. A counterexample for this transformation is $\{src.x.set \mapsto \lambda m.\ (m == 0 \vee m == 2),\ tgt.x.set \mapsto \lambda m.\ (m == 0 \vee m == 2),\ src.x.pc \mapsto False,\ tgt.x.pc \mapsto False\}$ which allows the target function to return the value 6 but not the source function.

3.6 Optimizations

3.6.1 Bounded Cardinality

In the scheme for encoding `undef` semantics in SMT described in Section 3.3, function arguments' set membership functions are encoded as uninterpreted functions with no constraints. This amounts to modeling these function arguments as having sets of values of cardinality up to the number of values permissible within their type. Such an encoding scheme that models both partial and full `undef` values is necessary in order to soundly guarantee that a transformation is valid. However, for transformations that are invalid, in the common case, we do not need to model sets of arbitrary cardinality. The insight is that in most cases, we can find a counterexample by modeling sets of bounded cardinality i.e. sets with at most 2 values, 3 values, etc. up to a fixed number of values.

Sets of bounded cardinality are realized by making the SMT solver find models for uninterpreted constants rather than uninterpreted functions. Set membership functions for function arguments with maximum cardinality k are encoded as follows:

$$\lambda x.(x == m_1) \vee (x == m_2) \vee \ldots (x == m_k)$$

Here, $m_1, m_2 \ldots m_k$ are the uninterpreted constants for which the SMT solver needs to find models as opposed to finding a model for an uninterpreted function.

This optimization is implemented in `superopt-refines` as an optional feature which the user can enable by passing as input, the bounded cardinality to be used for the sets of values of function arguments.

Chapter 4

Evaluation

4.1 Implementation

`superopt-refines` is implemented in C++. For evaluation, various pairs of programs in LLVM IR consisting of a single function per program containing no branches, loops, heap-manipulating instructions, or freeze instructions, and only integer types are examined. Symbolic executors have been implemented for LLVM IR, to convert the input programs to their corresponding transfer functions. The verification conditions for proving refinement of the input program pair are encoded as SMT expressions involving bitvectors, lambdas and uninterpreted functions. `superopt-refines` uses the Z3 SMT Solver [MB08] to solve the SMT proof obligations generated and these proof obligations are either over the AUFBV or UFBV theory. `superopt-refines` sets the time out parameter for the Z3 SMT Solver to be 60 seconds - if Z3 is unable to declare `sat` or `unsat` for an SMT expression in 60 seconds, it times out and the proof fails.

Threats to Validity. While `superopt-refines` tries to accurately reflect the semantics described in the LLVM Language Reference [Dev14], there could be unintended differences between the semantics formalized in `superopt-refines` and the semantics intended by the language developers.

4.2 Experimental Setup and Benchmark Selection

We evaluate and compare the results and execution times of `superopt-refines` and the online `alive-tv` tool (a bounded automatic translation validator for LLVM) [LLH+21] on: (1) transformations collected from [LMNR15], (2) transformations involving arithmetic operations over one to two variables. For each of the selected transformations, we attempt to prove that the target function refines the source function using `superopt-refines` and `alive-tv`.

4.3 Results

In tables 4.1 and 4.2, for brevity, the source and target functions are represented as the expressions that are returned by the respective functions. In the actual source and target

functions, the variables a and b are represented as bitvectors of width 32 (`i32`) in most cases, and in some cases, bitvectors of width 8 (`i8`). In tables 4.1 and 4.2, (1) `SR` Result and `AT` Result represent whether `superopt-refines` and `alive-tv` respectively declare the corresponding transformation to be valid or not, (2) `SR` Time and `AT` Time represent the time taken (in seconds) for `superopt-refines` and `alive-tv` to declare a result, and (3) `SR` Time (Bounded) in 4.2 represents the time taken by `superopt-refines` (in seconds) to declare a result with a bounded cardinality of 1 - 3 values for the function arguments — a and b.

Table 4.1 contains the results of evaluating `superopt-refines` and `alive-tv` on transformations that are valid as per LLVM semantics [Dev14]. For two transformations in Table 4.1, `superopt-refines` timed out and was unable to prove those transformations as valid. This happened due to SMT solver time-outs when verifying the condition ($tgt.ret\_val.pc \implies src.ret\_val.pc$) of the verification condition. In contrast, the `alive-tv` tool didn't time-out and was able to prove these transformations as valid. As for the remaining transformations, both `superopt-refines` and `alive-tv` were able to prove their validity. It is evident from the table that `superopt-refines` requires more time than `alive-tv` to prove validity of transformations. From the data given in Table 4.1, `superopt-refines` requires 11.84× more time than `alive-tv` on average. This slowdown and the cases in which `superopt-refines` timed out occur because `alive-tv` is a bounded translation validator and has made certain underapproximations in modeling LLVM semantics for efficiency purposes which allow it to declare results faster whereas `superopt-refines`'s modeling of LLVM semantics is intended to be sound. Ex. `alive-tv` makes the underapproximating assumption that function arguments are either well-defined values, `poison` or fully `undef` i.e. it doesn't model the case where function arguments can be partially `undef` [LLH+21]. `alive-tv` makes this underapproximation because a doubly exponential search space is spawned for each partially `undef` value and by making this underapproximation, it reduces the aforementioned search spaces to exponential. In contrast, `superopt-refines`'s goal is sound translation validation and thus, it has to also model the possibility of function arguments being partially `undef`. It does this using set membership functions as described in Section 3.3 and thus, has to search over doubly exponential search spaces (2^{2^n} for each n bit integer) for each possible partially `undef` value in the input programs.

Table 4.2 contains the results of evaluating `superopt-refines` and `alive-tv` on transformations that are invalid as per LLVM semantics. For one of the transformations in this table, `superopt-refines` times out but `alive-tv` doesn't. This is due to the same reason described for the time-outs in Table 4.1 — `alive-tv` has made some underapproximations for efficiency reasons which `superopt-refines` hasn't. As a result, `superopt-refines` has to search over a larger space to prove transformations which causes it to time-out for this transformation. From the remaining transformations in Table 4.2, the benefits of sound modeling of LLVM semantics in `superopt-refines` can be seen — `alive-tv` incorrectly

declares all but one of these transformations to be valid, whereas `superopt-refines` is able to correctly declare them as invalid. `alive-tv` incorrectly determines these transformations to be valid due to its underapproximating assumption that function arguments are either fully `undef` or not. For example, for the transformation $a+a \rightarrow a*3-a$, one possible counterexample is $\{a \mapsto \{0,2\}\}$ which allows the target program to return the value 6 but not the source program. `superopt-refines` can discover this counterexample but `alive-tv` cannot because it assumes variable a to be either not `undef` or fully `undef` (assuming variable a is of type `i8`, fully `undef` is $\{0,1,2\ldots255\}$).

Furthermore, it can be seen from comparing the columns `SR` Time and `SR` Time (Bounded) of Table 4.2, that the bounded cardinality optimization described in Section 3.6.1 helps `superopt-refines` disprove transformations faster. `superopt-refines` is able to disprove transformations $\sim 2\times$ faster on average with the bounded cardinality optimization.

Table 4.1: Valid Transformations

Source	Target	Correct Result	`AT` Result	`SR` Result	`AT` Time	`SR` Time
$(a \oplus -1)+b$	$b-1-a$	Valid	Valid	Valid	4.54	29.20
$a +_{nsw} 1 > a$	$True$	Valid	Valid	Timed Out	0.30	_
$-(a/5)$	$a/(-5)$	Valid	Valid	Timed Out	0.31	_
$a+a$	$a*2$	Valid	Valid	Valid	0.54	1.82
$a+a+a$	$a*2+a$	Valid	Valid	Valid	0.56	2.86
$a+a-a$	a	Valid	Valid	Valid	0.3	7.03
$(a+a)*a$	$a*a*2$	Valid	Valid	Valid	0.45	10.26
$a+a-a$	$a*2-a$	Valid	Valid	Valid	0.34	7.23
$(a+a)*2$	$a*4$	Valid	Valid	Valid	0.33	2.44
$a*2+a$	$a*3$	Valid	Valid	Valid	0.53	3.43
$a*2-4$	$(a-2)*2$	Valid	Valid	Valid	0.37	2.27
$(a+a)*b$	$a*2*b$	Valid	Valid	Valid	1.29	8.44
$a+a+b$	$a*2+b$	Valid	Valid	Valid	0.6	9.57

Table 4.2: Invalid Transformations

Source	Target	Correct Result	AT Result	SR Result	AT Time	SR Time	SR Time (Bounded)
$a -_{nsw} (-b)$	$a +_{nsw} b$	Invalid	Invalid	Timed Out	0.37	_	_
$a + a$	$a +_{nuw} a$	Invalid	Invalid	Invalid	0.30	4.09	0.43
a	$a + a - a$	Invalid	Valid	Invalid	0.30	0.93	1.08
a	$a * 2 - a$	Invalid	Valid	Invalid	0.36	1.09	0.98
$a + a$	$a * 3 - a$	Invalid	Valid	Invalid	0.51	4.18	1.59
$a - a$	$(a - a) * a$	Invalid	Valid	Invalid	0.55	2.63	1.18
$a * 3$	$a + a + a$	Invalid	Valid	Invalid	0.47	2.75	1.09
$a * 3$	$a * 2 + a$	Invalid	Valid	Invalid	0.45	4.25	1.83
$a * 3$	$a * 4 - a$	Invalid	Valid	Invalid	0.45	2.66	1.46
$a + a + a$	$a * 4 - a$	Invalid	Valid	Invalid	0.34	4.29	2.92
$(a - a) * a$	$(a - a) * 4$	Invalid	Valid	Invalid	0.41	6.94	1.68
$(a - a) * 3$	$a - a$	Invalid	Valid	Invalid	0.55	2.07	1.63
$(a - a) * 3$	$(a - a)/2$	Invalid	Valid	Invalid	0.50	3.82	2.89
$(a - a)/3$	$(a - a)/4$	Invalid	Valid	Invalid	0.56	7.69	5.30
$a * 2 - a$	$a + a - a$	Invalid	Valid	Invalid	0.31	1.39	2.57
$a * 3 + a$	$(a + a) * 2$	Invalid	Valid	Invalid	0.50	2.91	1.93
$a * 4 - a$	$a * 2 + a$	Invalid	Valid	Invalid	0.69	5.32	1.65
$a * 2 + a$	$a * 4 - a$	Invalid	Valid	Invalid	0.52	7.49	2.42
$a + b - a$	$b + b - b$	Invalid	Valid	Invalid	0.52	1.83	1.69
$a + b - a$	$b * 2 - b$	Invalid	Valid	Invalid	0.34	1.82	1.69

Chapter 5

Limitations and Future Work

In this chapter, we discuss the shortcomings of `superopt-refines`, and potential methods of addressing them.

5.1 Limited Support for LLVM IR

One of the limitations of `superopt-refines` is that it restricts the space of input programs it can accept by not allowing branches, loops, heap-manipulating instructions, and freeze instructions. In this section, we discuss how such instructions can be potentially modeled in `superopt-refines`,

5.1.1 Heap-manipulating Instructions

`superopt-refines` can be made to support heap-manipulating instructions by using SMT array theory to model the memory states of the source and target programs. The algorithm would track the effects of heap-manipulating instructions on the memory states of the source and target programs, and finally verify that the memory state of the target program refines the memory state of the source program.

5.1.2 `freeze` Instruction

`superopt-refines` can be made to support the `freeze` instruction by adding to the existing encoding scheme. The poison condition predicate for a `freeze` instruction would be $False$. The resulting set of values of a `freeze` instruction would be encoded in the following way,

Instruction:

```
%b = freeze T %a
```

Corresponding Set Membership Function:

`b.set` := $\lambda x.\ (\neg$`a.pc` $\implies$ `a.set(b.freeze_val)`$) \wedge (x ==$ `b.freeze_val`$)$

`b.freeze_val` is an uninterpreted constant of type corresponding to `T` in SMT, that represents the arbitrary, fixed value yielded by the `freeze` instruction. This set membership

function reflects that the `freeze` instruction yields a well-defined value through the term x == `b.freeze_val` which makes it a singleton set. Furthermore, the term (¬`a.pc` ⟹ `a.set(b.freeze_val)`) reflects that if register `%a` is not poison, the fixed value yielded by the `freeze` instruction must belong to the set of values yielded by register `%a`.

5.1.3 Branches and Loops

`superopt-refines` can be made to support branches and loops through bisimilarity checking. Bisimilarity checking proceeds by correlating transitions in the two input programs and identifying invariants between the variables of the two programs at the endpoints of the correlated transitions [PSS98]. As demonstrated in [GRB20, DB17a, DB17b], bisimilarity checking has been successful in statically computing proofs of equivalence between pairs of programs. The algorithm developed in [GRB20] can be extended to compute proofs of refinement between LLVM IR programs by extending the invariants inferred at the endpoints of the correlated transitions to also include refinement invariants (Section 3.4.1), and verifying that the return value and the memory state of the target program refine that of the source program.

5.2 SMT Solver Time-Outs

`superopt-refines` can fail to prove refinement for a pair of programs due to the SMT solver timing out. Certain instances of such failure are shown in Section 4.3 and the reasons for their occurrence are also discussed. While SMT solver time-outs do not affect soundness, they do affect the completeness of `superopt-refines` and this section discusses measures that can be taken to reduce the frequency of such time-outs.

5.2.1 Function Inlining and Simplification

`superopt-refines` relies on set membership functions to represent `undef` values and these functions are often invoked in the SMT proof obligation to indicate that a variable is a member of a set. However, relying on set membership functions needlessly obfuscates the SMT proof obligation and can hinder the SMT solver in using its strategies. It has been observed that `superopt-refines` can sometimes even time out on trivial queries such as $False \implies \ldots$. This suggests that simplifying the final SMT proof obligation by inlining all function invocations and removing all set membership functions from the proof obligation might help.

It turns out that inlining set membership functions isn't enough since resulting SMT proof

obligations still timed out for some benchmarks. From observing the generated SMT proof obligations of these sample benchmarks, it was hypothesized that the time-outs could be occurring due to the presence of too many existential quantifiers. `superopt-refines`'s encoding scheme for set membership functions, as discussed in Section 3.3, relies heavily on existential quantifiers. It was discovered from observing the SMT proof obligations of the sample benchmarks, that existential quantifiers can sometimes be redundant, and eliminated in such cases. Elimination of such redundant existential quantifiers combined with function inlining enabled the sample benchmarks to be proven by the SMT solver which were earlier causing it to time out. We speculate that function inlining combined with quantifier elimination and other such simplification heuristics can not only allow more proof obligations to become decidable by the SMT solver, but also allow the SMT solver to decide proof obligations faster. Below, a sanitized snippet of a proof obligation is shown (in pseudocode), that was initially causing the SMT solver to time out but after function inlining and simplification, allowed the SMT solver to prove it successfully.

Snippet of Original Proof Obligation:

let $f_1 = \lambda m.\ m == src.x.pc$ in
let $f_2 = \lambda m.\ m == dst.x.pc$ in
$\exists\ x_1, x_2.\ f_1(x_1) \wedge f_2(x_2) \wedge (x_2 \implies x_1)$

After Function Inlining:

$\exists\ x_1, x_2.\ (x_1 == src.x.pc) \wedge (x_2 == dst.x.pc) \wedge (x_2 \implies x_1)$

After Function Inlining and Simplification:

$\exists\ x_1, x_2.\ (dst.x.pc \implies src.x.pc)$

After Function Inlining, Simplification, and Redundant Quantifier Elimination:

$dst.x.pc \implies src.x.pc$

5.2.2 Query Decomposition

Previous work [GSMB18] has shown that the technique of query decomposition can help decide proof obligations that the SMT solver otherwise can't decide, and can also yield gains in the time taken to decide proof obligations. Query decomposition decomposes larger expressions into smaller sub-expressions, and substitutes those sub-expressions that are equivalent. Similar techniques can also be used for inferring invariants for refinement as described henceforth.

Theorem 1. Let a, b, c, d be program variables/registers. Then we have the following property,

$$\frac{a \sqsupseteq_u c \qquad b \sqsupseteq_u d}{a \; op \; b \sqsupseteq_u c \; op \; d}$$

Proof. Take an arbitrary value v that belongs to the set of values of $c \; op \; d$. Then, there exist c_1 and d_1 belonging to the set of values of c and d respectively such that $v = c_1 \; op \; d_1$. Since $a \sqsupseteq_u c$ and $b \sqsupseteq_u d$, there exist a_1 and b_1 belonging to the set of values of a and b respectively such that $a_1 = c_1$ and $b_1 = d_1$. Since $c_1 \; op \; d_1 = a_1 \; op \; b_1 = v$, any value v that belongs to the set of values of $c \; op \; d$ also belongs to the set of values of $a \; op \; b$. QED.

This result can be generalized for arbitrary expressions using structural induction.

Usage. We can use this theorem to infer new invariants without the use of an SMT solver. For example, using the invariants $a \sqsupseteq_u a', b \sqsupseteq_u b', c \sqsupseteq_u c', d \sqsupseteq_u d'$, we can infer the invariant $a + b * c/d \sqsupseteq_u a' + b' * c'/d'$ without using an SMT solver.

It is also possible to use this theorem to make invariants simpler to prove for SMT solvers by replacing isomorphic sub-expressions with a single variable in source and target expressions and adding an invariant in the precondition for refinement of the newly added variables. For example, the query

$$a \sqsupseteq_u a', b \sqsupseteq_u b', c \sqsupseteq_u c', d \sqsupseteq_u d' \implies a + b * c/d \sqsupseteq_u a' - b' * c'/d'$$

can be converted to the following simpler query,

$$a \sqsupseteq_u a', t \sqsupseteq_u t' \implies a + t \sqsupseteq_u a' - t'$$

As per Theorem 1, the results of both SMT queries would be equivalent.

Chapter 6

Discussion and Conclusion

Translation validation attempts to certify optimizers and compilers by certifying individual executions. Translation validation is undecidable in general and yet, is a relevant problem in many fields such as program synthesis, superoptimization, etc.

To explore certifying the LLVM optimizer using translation validation methods, we have created and described an automatic sound translation validation prototype — `superopt-refines`. Through `superopt-refines`, we explore the problems and benefits of sound modeling of LLVM IR semantics. We've evaluated `superopt-refines` against `alive-tv` (a bounded automatic translation validator for LLVM) and shown 12 transformations where `superopt-refines` and `alive-tv` yield the correct result, 18 transformations where `superopt-refines` yields the correct result and `alive-tv` yields the incorrect result, and 3 transformations where `alive-tv` yields the correct result and `superopt-refines` times out. In this evaluation, `superopt-refines`'s detection of incorrect transformations which `alive-tv` fails to detect demonstrates the benefits of sound modeling of LLVM IR semantics. However, this evaluation also demonstrates the shortcomings of `superopt-refines` through the transformations where it has timed out. Future work in `superopt-refines` would attempt to extend support for more LLVM IR language features, and reduce frequency of time-outs by optimizing the SMT encoding and required proof obligations.

Bibliography

[DB17a] Manjeet Dahiya and Sorav Bansal. Black-box equivalence checking across compiler optimizations. In *Asian Symposium on Programming Languages and Systems*, pages 127–147. Springer, 2017.

[DB17b] Manjeet Dahiya and Sorav Bansal. Modeling undefined behaviour semantics for checking equivalence across compiler optimizations. In *Haifa Verification Conference*, pages 19–34. Springer, 2017.

[Dev14] LLVM Developers. Llvm language reference manual. https://llvm.org/docs/LangRef.html, 2014. Accessed: 2021-12-31.

[GRB20] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. Counterexample-guided correlation algorithm for translation validation. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[GSMB18] Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. Effective use of smt solvers for program equivalence checking through invariant-sketching and query-decomposition. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 365–382. Springer, 2018.

[LKS+17] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P Lopes. Taming undefined behavior in llvm. *ACM SIGPLAN Notices*, 52(6):633–647, 2017.

[LLH+21] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021.

[LMNR15] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 22–32, New York, NY, USA, 2015. Association for Computing Machinery.

[MB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166. Springer, 1998.