creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사학위논문

컴파일러의 부동소수점 최적화 검증을 위한 효율적인 SMT 인코딩

An Efficient SMT Encoding for the Validation of Floating-Point Compiler Optimizations

2023 년 2 월

서울대학교 대학원

컴퓨터 공학부

남 승 현

공학석사학위논문

컴파일러의 부동소수점 최적화 검증을 위한 효율적인 SMT 인코딩

An Efficient SMT Encoding for the Validation of Floating-Point Compiler Optimizations

2023 년 2 월

서울대학교 대학원

컴퓨터 공학부

남 승 현

컴파일러의 부동소수점 최적화 검증을 위한 효율적인 SMT 인코딩

An Efficient SMT Encoding for the Validation of Floating-Point Compiler Optimizations

지도교수 허 충 길

이 논문을 공학석사학위논문으로 제출함

2023 년 2 월

서울대학교 대학원

컴퓨터 공학부

남 승 현

남승현의 석사학위논문을 인준함

2023 년 2 월

위 원 장	이 광 근	(인)
부위원장	허 충 길	(인)
위 원	권 태 경	(인)

요약

부동소수점 값을 추상화한 비트벡터와 연산을 추상화한 SMT 수식을 이용해 컴파일러 최적화를 빠르고 정확하게 검증하는 방법을 소개하고, 실험을 통해 효용성을 확인한다. IEEE-754 부동소수점 표준에 정의된 다수의 부동소수점 연산을 머신러닝 애플리케이션 최적화 검증에 사용할 수 있을 정도로 정확하게 상위 근사(overapproximation) 하는 방법을 고안하였다. 추상화된 연산은 표준 부동소수점 연산에 비해 절대적인 연산 횟수도 적고 SMT solver가 더욱 효율적으로 검증할 수 있다. 머신러닝 컴파일러 프레임워크인 MLIR에서 사용되는 최적화를 표준 부동소수점 연산을 그대로 재현하는 방법과 제시하는 방법으로 검증해 비교했다. 동일한 최적화를 약 14배 더 빠르게 검증했고 timeout이 줄어들며 더 많은 최적화를 검증하였다. 이 과정에서 잘못 구현된 MLIR 최적화를 발견해 제보하였고 현재는 제거되었다.

주요어: 최적화 검증, SMT solver, 부동소수점

학번: 2021-22699

목차

요약 **i**

목차 **ii**

그림 목차 **iv**

표 목차 **v**

제 1 장 서론 **1**

제 2 장 배경 지식 **2**

2.1 SMT Solver 2

2.2 IEEE-754 표준 2

2.3 부동소수점 연산의 검증 2

2.4 MLIR 프레임워크 3

2.5 MLIR-TV 3

제 3 장 프로그램 추상화와 부동소수점 연산의 검증 **4**

3.1 프로그램 추상화를 이용한 프로그램 검증 4

3.2 프로그램 추상화 방법 4

3.2.1 상위 근사 4

3.2.2 하위 근사 5

제 4 장 인코딩 규칙과 구현 **6**

4.1 기본 구조 6

4.2 연산의 구현 7

4.2.1 부호 반전 및 절댓값 연산 7
4.2.2 대소 비교 7
4.2.3 새로운 값을 내놓는 연산 8
4.3 상수의 값 반영 8
4.3.1 검증에 필수적인 상수의 존재 보장 8
4.3.2 항등원과의 연산 9
4.3.3 특수한 상수와의 연산 9
4.3.4 상수 간의 대소관계 10
4.3.5 상수 사이에 올 수 있는 값의 개수 10
4.4 비트벡터의 너비 정하기 11

제 5 장 정밀도 변환을 지원하는 인코딩 규칙 12
5.1 기본 구조 12
5.2 연산의 구현 14
5.2.1 높은 정밀도로 형변환 14
5.2.2 낮은 정밀도로 형변환 14
5.3 상수의 값 반영 16
5.4 비트벡터의 너비 정하기 16

제 6 장 최적화 검증 및 성능 비교 18
6.1 실험 결과 비교 18
6.2 탐지한 잘못된 최적화 20

참고문헌 21

Abstract 22

그림 목차

그림 4.1 비트벡터 인코딩의 기본 구조 6

그림 5.1 확장된 비트벡터 인코딩의 구조 13

표 목차

표 4.1 비트벡터 인코딩 설명에 사용되는 기호 7

표 5.1 확장된 비트벡터 인코딩 설명에 사용되는 기호 13

표 6.1 비트벡터 인코딩과 FPA 인코딩을 이용해 최적화를 검증한 결과 19

제 1 장 서론

본 논문에서는 컴파일러 최적화를 합리적인 시간 내에 검증하기 위해 SMT solver에서 부동소수점 논리 (FPA logic) 대신 활용할 수 있는 인코딩을 제시하고, 실험을 통해 실제 효용성을 확인한다.

근래 머신 러닝 애플리케이션에서 수행하는 연산이 점점 더 복잡해지는 한편 가속기의 성능이 기하급수적으로 발전함에 따라, 프로그래머가 고수준의 텐서 연산으로 정의한 프로그램을 가속기를 활용하는 최적화된 기계어로 변환해주는 머신 러닝 컴파일러의 중요성이 점점 커지고 있다. 자연스레 머신 러닝 컴파일러 최적화의 검증 또한 중요한 문제로 떠올랐는데, 머신 러닝 애플리케이션이 많은 양의 부동소수점 텐서를 다룬다는 점은 최적화를 검증하는 데에 있어서 걸림돌로 작용한다.

부동소수점 연산 위주의 프로그램을 검증하는 것은 정수 연산 위주의 프로그램을 검증하는 것보다 본질적으로 어렵다. 부동소수점 연산은 정수에 비해 복잡하게 정의되어 있고, 이로 인해 연산 시간이 늘어나게 되면 검증에 소요되는 시간도 같이 늘어나게 된다. 머신 러닝 애플리케이션은 원소가 수백만 개에 달하는 텐서를 다루기도 한다는 점을 고려했을 때 개별 연산을 검증하는 시간이 증가한다면 프로그램 전체를 합리적인 시간 내에 검증하는 것은 사실상 불가능하게 된다.

부동소수점 연산의 복잡도는 대부분 새로운 값을 정확하게 표현하기 위한 과정에서 온다. 하지만 머신 러닝 컴파일러의 최적화는 새로운 값을 계산하기보다 연산 순서나 메모리 상 값의 위치를 바꾸는 경우가 많으며, 이를 검증하는 것은 두 값이나 텐서를 비교한 결과가 이전과 동일하다는 것을 보이는 것으로 충분하다. 이 사실에 착안해 고안한 인코딩으로 부동소수점 텐서 연산이 포함된 최적화를 검증한 결과 부동소수점 논리를 활용했을 때에 비해 훨씬 빠르게 검증했으며, 컴파일러 최적화에 존재하던 버그를 제보해 현재는 제거되었다.

제 2 장 배경 지식

2.1 SMT Solver

SMT는 Satisfiability Modulo Theory의 줄임말로, SMT solver는 SMT 수식을 SAT 수식으로 바꾸어 조건을 만족시키는 입력값이 존재하는지 탐색한다. SMT solver의 탐색 결과는 3가지로, 조건을 만족시키는 입력값이 존재하는 경우 (SAT), 만족시키는 입력값이 존재하지 않는 경우 (UNSAT), 그리고 제한된 시간 내에 입력값의 존재 유무를 판별하지 못한 경우 (timeout)이다.

2.2 IEEE-754 표준

IEEE-754 표준[7]은 컴퓨터에서 고정 비트를 이용해 부동소수점 방식으로 실수를 표현하기 위한 표준적인 방법으로, 정해진 너비 (32비트, 64비트 등)의 비트를 부호부, 지수부, 가수부로 나누어 어떤 실수를 부호, 지수, 가수의 곱을 통해 표현한다. 사칙연산, 나머지, 정수로의 변환, 정밀도 변환 등의 연산을 지원한다.

2.3 부동소수점 연산의 검증

현재 주로 쓰이는 SMT solver들은 정수나 비트벡터에 관련된 논리 (logic) 뿐만 아니라 IEEE-754에 규정된 부동소수점 연산에 대한 논리 (FPA logic)[8] 또한 지원하며, 이를 이용하면 프로그램에 등장하는 부동소수점 연산도 정확하게 검증할 수 있다.

하지만 부동소수점 연산은 정수 연산에 비해 매우 복잡하다. 예시로 가장 간단한 부동소수점 더하기는 exponent 빼기 (subtraction) - mantissa 시프트 (shift) - mantissa 더하기 (addition) - exponent 증가 (increment)의 4단계로 이루어지며, 부동소수점 곱하기, 나누기, 조이기 (clamp), 정밀도 변환 등은 더 많은 과정을 거

친다. 연산이 복잡해지면 연산을 수행하는 데 걸리는 시간이 늘어나고, 검증에도 더 많은 시간을 소요하게 된다. 이는 실질적으로 검증 가능한 프로그램을 제약하여 검증의 범용성과 유용성을 떨어트린다.

2.4 MLIR 프레임워크

MLIR[1]은 Multi-Level Intermediate Representation의 약자로, 머신 러닝 애플리케이션에 사용되는 고수준의 operation부터 저수준의 하드웨어 레벨 최적화까지 다루기 위한 컴파일러 프레임워크이다.

다양한 수준의 operation을 다루기 위해 MLIR은 사용자가 새로운 중간언어(IR) 를 정의, 활용, 변환, 조합할 수 있도록 하며, 이런 중간언어를 dialect라고 한다[3]. Dialect는 특정 목적을 가진 중간언어 부분집합으로, 각 dialect에는 새로운 연산과 타입을 정의할 수 있다. 예를 들어 tensor dialect[4]는 tensor 타입과 tensor를 다루기 위한 연산을 제공하며, TOSA (Tensor Operator Set Architecture) dialect[5]는 tensor 간의 연산을 제공한다. Dialect는 단순 연산 뿐만 아니라 프로그램의 control flow나 더 저수준의 하드웨어 조작도 지원한다.

MLIR 프로그램은 여러 단계에 걸친 dialect들의 조합으로 이루어지며, MLIR의 최적화 패스는 다양한 dialect 간 변환을 지원한다. 고수준의 자료구조와 연산을 저수준의 자료구조와 연산으로, 그리고 다시 하드웨어 명령어로 점차 내리는(lowering) 것이 MLIR 최적화 패스의 기본 골자이다.

2.5 MLIR-TV

MLIR-TV[2]는 MLIR 프로그램 최적화 검증 툴로, 어떤 최적화를 적용하기 전 원본 MLIR 함수와 최적화를 적용한 함수를 입력으로 받아 SMT 수식으로 변환한다. 그 뒤 SMT solver를 이용해 두 수식에 동일한 값을 대입했을 때 출력값이 달라지는 경우가 존재하는지 탐색한다. 만약 SMT solver에서 수식의 출력값이 달라지는 입력값을 찾아낸다면 틀린 최적화, 해당하는 입력값이 없음을 보인다면 옳은 최적화로 판별한다.

제 3 장 프로그램 추상화와 부동소수점 연산의 검증

이번 장에서는 본 연구의 이론적 기반인 프로그램 추상화와 프로그램 검증에 대해 설명한다.

3.1 프로그램 추상화를 이용한 프로그램 검증

프로그램을 검증하는 데는 프로그램을 어느 정도 추상화 (abstraction) 한 모델을 활용하는 방법이 자주 쓰인다. 프로그램을 추상화하지 않고 행위 (behavior) 를 동일하게 재현한 모델을 검증하는 것은 가장 정확하지만, 모델의 크기, 즉 검증해야 할 상태의 수가 과도하게 크기 때문에 적절한 시간 내에 검증하기 어렵다는 문제가 있다. 반대로 프로그램을 과도하게 추상화하면 프로그램이 가진 행위와 모델이 가진 행위가 크게 달라져 검증의 정확도가 떨어진다. 따라서 검증 시간과 정확도 사이의 적절한 타협점을 찾는 것이 추상화를 이용한 프로그램 검증의 핵심이다.

3.2 프로그램 추상화 방법

프로그램을 추상화하는 방법에는 상위 근사 (over-approximation) 와 하위 근사 (under-approximation) 가 있다.

3.2.1 상위 근사

상위 근사는 원본의 행위를 포함해 더 많은 행위를 가지는 모델을 생성한다. 원본의 행위는 추상화한 모델을 벗어나지 않으므로, 상위 근사한 모델이 검증된다면 (검증된 행위만을 가질 수 있는) 원본 또한 올바르다는 것을 알 수 있다. 그러나 모델 검증에 실패했을 때 그것이 원본에 존재하는 행위 때문인지, 원본에는 없지만 근사한 모델에만 존재하는 행위 때문인지는 알 수 없다.

3.2.2 하위 근사

하위 근사는 상위 근사와 반대로 원본의 행위를 일부 잃어 더 적은 행위를 가지는 모델을 생성한다. 원본의 행위는 추상화한 모델의 행위를 (올바르지 않은 행위를 포함해) 전부 포섭하므로, 하위 근사한 모델이 검증 실패한다면 원본 또한 올바르지 않음을 알 수 있다. 하지만 근사한 모델의 검증에 성공했다 하더라도 원본에는 있지만 모델에는 존재하지 않아 놓친 행위가 있기 때문에 원본을 완벽하게 검증했다고 할 수 없다.

제 4 장 인코딩 규칙과 구현

이번 장에서는 3장에서 설명한 부동소수점 검증의 문제를 해결하는 새로운 인코딩을 제시한다. 간결하고 명확한 설명을 위해 필요한 경우에는 표 4.1의 기호로 대신 표기한다.

4.1 기본 구조

제시하는 인코딩은 대다수 SMT Solver에서 지원하는 비트벡터 (Bit Vector)를 기반으로 한다. 비트벡터의 너비 (Bitwidth)는 특정 값으로 고정되어 있지 않으며, 검증하려는 프로그램에 따라 달라진다. 프로그램 내에서 여러 정밀도의 부동소수점 변수가 등장한다면 정밀도에 따라 서로 다른 길이의 비트벡터로 표현된다.

그림 4.1 비트벡터 인코딩의 기본 구조

어떤 부동소수점 값을 인코딩한 비트벡터는 다음과 같이 해석한다.

- 비트벡터의 가장 앞 비트 (MSB) 는 부호 영역 (Sign Bit, SB) 이며, SB가 0이라면 양의, 1이라면 음의 부동소수점 값을 의미한다.
- 비트벡터의 SB를 제외한 비트들은 크기 영역 (Magnitude Bits, MB) 이며, 프로그램에 등장하는 부동소수점 값의 절댓값 정보를 담고 있다.
  - 서로 다른 MB는 반드시 서로 다른 절댓값을 의미한다.
  - 실수 절댓값 사이의 대소관계는 해당 값을 의미하는 MB 사이에도 동일하게 적용된다.

SB	부호 비트
MB	크기 비트
P_{src}	최적화를 적용하기 전 원본 프로그램
P_{tgt}	최적화를 적용한 검증 대상 프로그램
$P_{(src,tgt)}$	검증에 사용되는 P_{src}와 P_{tgt}쌍
f_i $(i \in \mathbb{N})$	IEEE-754에 정의된 i비트 부동소수점 타입
$\#\mathrm{f}_i$ $(i \in \mathbb{N})$	$P_{(src,tgt)}$에 등장하는 i비트 부동소수점 타입 변수의 개수
BV_i $(i \in \mathbb{N})$	i비트 부동소수점 값을 인코딩하기 위한 비트벡터의 타입
$\lvert\mathrm{BV}\rvert$	비트벡터 BV의 너비 (비트수, bitwidth)

표 4.1 비트벡터 인코딩 설명에 사용되는 기호

- MB의 모든 비트가 0이라면 절댓값 0.0을 의미한다.
- 최하위 비트 (LSB) 를 제외한 모든 비트가 1이라면 Inf, 최하위 비트도 1이라면 NaN을 의미한다.
- 위에서 언급한 경우를 제외하면 MB가 의미하는 구체적인 값은 검증 대상에 따라 달라진다.

4.2 연산의 구현

4.2.1 부호 반전 및 절댓값 연산

SB를 반전(flip)시킴으로써 양수를 음수로, 음수를 양수로 변환할 수 있으며, SB에 0을 대입하면 어떤 수에 절댓값 연산을 적용한 것과 같다.

4.2.2 대소 비교

두 비트벡터에서 SB가 서로 다르다면 $SB = 0$ 인 비트벡터 (양수) 가 $SB = 1$ 인 비트벡터 (음수) 보다 항상 크다.

두 비트벡터의 SB가 서로 동일하다면 값의 절댓값을 표현하는 MB를 이용해 비교한다. MB가 동일하다면 둘은 같은 값이며, MB가 서로 다르고 $SB = 0$ 이라면 MB의 비교 결과, $SB = 1$ 이라면 MB의 비교 결과와 반대이다.

단, 두 비트벡터 중 하나 이상이 NaN라면 비교 결과는 IEEE-754 표준에 따라 항상 false이다.

4.2.3 새로운 값을 내놓는 연산

사칙연산 등 새로운 값을 내놓는 연산은 후술할 일부 예외를 제외하면 각각을 서로 다른 미해석 함수 (Uninterpreted Function; UF) 로 처리한다.

연산을 UF로 모델링하는 것은 구체적인 연산을 상위 근사하는 것이다. SMT solver는 본래 연산의 것을 포함한 모든 함수의 정의 중에서 $P_{(src,tgt)}$의 실행이 서로 달라지는 정의가 존재하는지 탐색하는데, 만약 그런 함수가 존재하지 않는다면 상위 근사의 특성에 의해 본래 연산의 정의를 따르더라도 $P_{(src,tgt)}$의 실행이 동등함이 보장되는 것이다.

단, 가환적인 (commutative) 연산에서 피연산자의 순서만 바뀐 경우는 항상 같은 값을 반환하도록 별도의 예외 처리를 해주었다.

4.3 상수의 값 반영

프로그램에 상수가 등장하는 경우에는 상수의 특성이나 상수 사이의 관계가 검증 과정에 제대로 반영되도록 해야 한다. 이를 정확하게 반영하지 않으면 SMT solver가 잘못된 반례를 찾아내 프로그램의 검증에 실패할 가능성이 높아져서 검증의 유용성이 크게 떨어질 수 있기 때문이다.

4.3.1 검증에 필수적인 상수의 존재 보장

0.0, 1.0, Inf, NaN은 부동소수점이 등장하는 프로그램의 검증에 반드시 필요한 상수이다. 이 상수들은 IEEE-754 표준에 특별한 연산이 정의되어 있거나 특정 연산의 항등원인 상수로, 해당 연산들을 검증하기 위한 예외 처리에 사용되므로

이 상수들을 인코딩한 비트벡터가 반드시 존재해야 한다.

이 중 0.0, Inf, NaN의 비트벡터 인코딩은 4.1절 에 따라 정해져 있으므로 별다른 작업이 필요 없지만, 1.0의 비트벡터는 정해져 있지 않으므로 프로그램에 1.0이 직접적으로 등장하지 않으면 예외 처리에 사용할 비트벡터를 알 수 없게 된다. 따라서 실제 프로그램에 존재하지 않더라도 1.0은 반드시 존재한다고 간주하고, 후술할 과정을 통해 비트벡터를 할당해 주어야 한다.

4.3.2 항등원과의 연산

일부 연산은 IEEE-754 표준에 따른 항등원이 존재하는데, 이때는 다른 쪽 피연산자를 그대로 반환해야 한다. IEEE-754 표준에 따른 항등원은 다음과 같다.

- 더하기: -0.0
- 빼기: 0.0
- 곱하기, 나누기: 1.0

피연산자 중 어느 한쪽이 해당 연산의 항등원인 경우 UF에 값을 대입하는 대신 반대쪽 피연산자를 반환하도록 하는 SMT 조건부 수식 (if-then-else; ite) 을 사용하면 항등원과의 연산을 구현할 수 있다.

4.3.3 특수한 상수와의 연산

IEEE-754 표준에 따른 0.0, Inf, NaN관련 예외적인 연산의 결과는 다음과 같다. (피연산자 순서 무관)

- 사칙연산에서 피연산자 중 하나 이상이 NaN $\rightarrow$ NaN
- $-\text{Inf} + \text{Inf} = \text{NaN}$
- $\pm\text{Inf} + \text{finite } x = \pm\text{Inf}$
- $\pm 0.0 \times \pm\text{finite } x = \pm 0.0$

- $\pm 0.0 \times \pm \mathrm{Inf} = \mathrm{NaN}$

항등원과의 연산과 마찬가지로 UF에 값을 대입하는 대신 특정 값을 반환하도록 하는 SMT 조건부 수식을 사용하면 특수한 상수와의 연산을 구현할 수 있다.

4.3.4 상수 간의 대소관계

$P_{(src,tgt)}$에 등장하는 상수를 각각 임의의 비트벡터에 대응시키면 값을 인코딩한 비트벡터 사이 대소관계와 실제 값 사이 대소관계가 일치하지 않게 된다. 이 경우 SMT solver가 상수와 값을 비교하는 프로그램에서 실존하지 않는 반례를 찾아 검증에 실패할 수 있다. 따라서 검증의 정확도를 높이기 위해서는 실제 상수 사이의 관계를 상응하는 비트벡터에 반영해주어야 한다.

먼저 $P_{(src,tgt)}$에 존재하는 특정 정밀도의 상수를 모두 모은 뒤, 모은 상수에 전부 절댓값을 취해서 정렬한다. 각 상수마다 너비가 $|\mathrm{BV}_i|-1$, 즉 $|MB|$인 비트벡터 변수를 생성한 뒤, 상수 기준으로 정렬했을 때 인접한 모든 변수 쌍마다 실제 절댓값의 대소관계를 검증에 사용할 precondition으로 넣어준다. 이 과정을 $P_{(src,tgt)}$에 존재하는 모든 정밀도 i에 대해 반복하면 실제 상수의 대소관계를 검증에 정확하게 반영할 수 있다.

4.3.5 상수 사이에 올 수 있는 값의 개수

부동소수점 체계에서 두 상수 사이에 존재하는 값의 개수는 한정되어 있다. $P_{(src,tgt)}$에 등장하는 두 상수 사이에 존재하는 값의 개수보다 상수를 인코딩한 비트벡터 사이에 존재하는 값의 개수가 더 많다면 검증 과정에서 실존하지 않는 반례를 찾아 검증에 실패할 수 있다. 따라서 검증의 정확도를 높이기 위해서는 각 비트벡터 사이에 존재하는 값의 최대 개수를 실제 상수 사이에 존재하는 값의 개수로 제한해주어야 한다.

4.3.4에서 생성한 각 상수를 나타내는 비트벡터 변수에 대해서 비트벡터 값의 차이를 실제 절댓값 사이 값의 개수로 제한하는 SMT 수식을 검증에 사용할

precondition으로 넣어준다. 이 과정을 마찬가지로 $P_{(src,tgt)}$에 존재하는 모든 정밀도에 대해 반복하면 상수 사이 관계를 검증에 좀 더 정확하게 반영할 수 있다.

4.4 비트벡터의 너비 정하기

부동소수점 값을 인코딩하는 데 사용하는 비트벡터의 너비는 프로그램에서 등장하는 변수가 가질 수 있는 값의 가짓수를 제한한다. 만약 인코딩에 사용한 비트벡터의 너비가 너무 좁아 값의 가짓수가 변수의 개수보다 적어진다면 비둘기집의 원리에 의해 일부 변수는 반드시 중복된 값을 가지게 된다. 이 경우 프로그램에서 발생할 수 있는 모든 경우의 수를 표현할 수 없으므로 검증의 정확도가 떨어지게 된다. 반면 비트벡터의 너비가 너무 넓으면 검증의 정확도에는 영향을 주지 않지만 SMT solver의 연산 부담이 늘어나게 된다.

프로그램에 부동소수점의 정밀도 변환 (extend, truncate) 이 존재하지 않는다면 서로 다른 정밀도의 부동소수점 변수 사이에는 어떠한 관계도 성립하지 않는다고 가정할 수 있다. 이 경우 어떤 $N \in \mathbb{N}$에 대해 BV_N은 최소 $2 \times (\#\mathrm{f}_N + 4)$ 가지 값을 표현할 수 있어야 하는데, 이는 검증 과정에서 $0.0, 1.0$, Inf, NaN은 반드시 필요하고, 모든 값에 대해 절댓값이 같고 부호가 반대인 값도 존재하기 때문이다. 실제 검증에 사용하는 $|\mathrm{BV}_N|$은 이를 만족하는 최소한의 너비로 정하면 된다.

제 5 장 정밀도 변환을 지원하는 인코딩 규칙

IEEE-754 표준에는 두 가지 부동소수점 정밀도 변환이 정의되어 있는데, 하나는 어떤 부동소수점 값을 더 높은 정밀도로 변환하는 것이고 (extend), 다른 하나는 어떤 부동소수점 값을 더 낮은 정밀도로 변환하는 것이다 (truncate). 하지만 4장에서 정의한 인코딩 규칙에선 서로 다른 정밀도의 부동소수점 값 사이에 어떤 관계도 없다고 간주하기 때문에 정밀도 변환을 제대로 정의할 수 없다.

이번 장에서는 부동소수점 정밀도 변환이 사용되는 프로그램을 검증하기 위해 4장에서 정의한 인코딩 규칙을 확장하는 방법을 제시한다. 간결하고 명확한 설명을 위해 필요한 경우에는 표 5.1의 기호로 대신 표기한다.

5.1 기본 구조

프로그램에 정밀도 간 변환이 존재하는 경우를 검증할 때는 비트벡터를 해석할 때 4.1절에 정의된 조건 외에도 다음을 추가적으로 만족한다고 가정한다.

- MB는 최상위 비트 (MSB) 부터 확장 비트 (Extension Bits, EB), 내림 비트 (Floored Bits, FB), 오프셋 비트 (Offset Bits, OB) 의 3개 영역으로 나누어지며, 각 영역의 너비는 $P_{(src,tgt)}$에 따라 달라진다. 단, $P_{(src,tgt)}$에 등장하는 가장 낮은 정밀도의 부동소수점 타입에는 EB와 OB가 존재하지 않는다.

- $P_{(src,tgt)}$에 대해 BV_i의 $|FB|$는 $\mathrm{BV}_{\lfloor i \rfloor}$의 $|MB|$와 동일하다.

- $P_{(src,tgt)}$의 어떤 BV_i에 대해 BV_i 비트벡터의 EB=0이라면 이 비트벡터는 $\mathrm{f}_{\lfloor i \rfloor}$에서 표현 가능한 최댓값 이하의 실수를 의미하며, 0이 아니라면 그보다 큰 값을 의미한다.

- $P_{(src,tgt)}$의 어떤 BV_i에 대해 BV_i 비트벡터의 EB=0일 때, FB는 $\mathrm{f}_{\lfloor i \rfloor}$로 내림한 실수를 $\mathrm{BV}_{\lfloor i \rfloor}$으로 인코딩한 비트벡터의 MB와 동일하다.

EB	확장 비트
FB	내림 비트
OB	오프셋 비트
$\mathrm{f}_{\lfloor i \rfloor}$	$P_{(src,tgt)}$에서 f_i 다음으로 높은 정밀도의 IEEE-754 부동소수점 타입
$\mathrm{f}_{\lceil i \rceil}$	$P_{(src,tgt)}$에서 f_i 다음으로 낮은 정밀도의 IEEE-754 부동소수점 타입
$\mathrm{BV}_{\lfloor i \rfloor}$	$\mathrm{f}_{\lfloor i \rfloor}$ 값을 인코딩하기 위한 비트벡터의 타입
$\mathrm{BV}_{\lceil i \rceil}$	$\mathrm{f}_{\lceil i \rceil}$ 값을 인코딩하기 위한 비트벡터의 타입

표 5.1 확장된 비트벡터 인코딩 설명에 사용되는 기호

- $P_{(src,tgt)}$의 어떤 BV_i에 대해 BV_i 비트벡터의 EB=0일 때, OB=0이라면 이 비트벡터는 $\mathrm{f}_{\lfloor i \rfloor}$에서 정확하게 표현 가능한 실수를 의미하며, 0이 아니라면 정확하게 표현할 수 없는 실수를 의미한다.

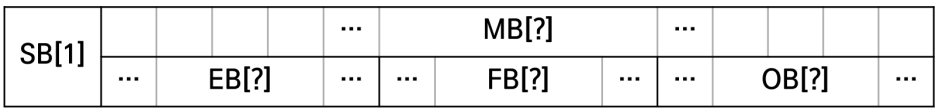

그림 5.1 확장된 비트벡터 인코딩의 구조

위 규칙에 따라 나눈 EB, FB, OB는 다음과 같이 해석한다.

- 만약 이 실수가 $\mathrm{f}_{\lfloor i \rfloor}$에서 표현 가능한 범위 내라면, 즉 EB=0이라면 FB는 MB가 의미하는 실수를 $\mathrm{f}_{\lfloor i \rfloor}$로 내림했을 때 어떤 값이 되는지를 의미하며, OB는 MB가 의미하는 값과 FB가 $\mathrm{f}_{\lfloor i \rfloor}$에서 의미하는 값의 차를 의미한다.

- 만약 이 실수가 $\mathrm{f}_{\lfloor i \rfloor}$에서 표현 가능한 범위를 벗어나는 값이라면, 즉 $EB \neq 0$이라면 EB, FB, OB를 모두 이어붙여 $\mathrm{f}_{\lfloor i \rfloor}$에서 표현 불가능한 어떤 값의 MB로 해석한다.

5.2 연산의 구현

5.2.1 높은 정밀도로 형변환

확장된 인코딩에서는 정해진 규칙에 따라 임의의 BV_i 비트벡터를 $\mathrm{BV}_{\lceil i \rceil}$ 비트벡터로 변환할 수 있다. 정밀도 변환을 거쳐도 값의 부호, 즉 SB는 동일하므로, BV_i 비트벡터의 MB를 $\mathrm{BV}_{\lceil i \rceil}$의 MB로 변환한 뒤 BV_i 비트벡터의 SB를 앞에 붙여주면 $\mathrm{BV}_{\lceil i \rceil}$ 비트벡터가 된다.

이때 변환 과정에서 BV_i 비트벡터의 MB가 $\mathrm{BV}_{\lceil i \rceil}$ 비트벡터의 FB와 달라지게 된다면 앞서 정의한 규칙에 어긋나게 된다. 이로 인해 실제로는 올바른 최적화도 검증에 실패할 수 있다. 따라서 변환 후에도 해당 비트들이 동일한 값을 가지도록 정확한 변환 규칙을 정해주어야 한다.

조건을 만족시키며 BV_i 비트벡터의 MB를 $\mathrm{BV}_{\lceil i \rceil}$의 MB로 변환하는 방법은 다음과 같다.

- 변환하는 값이 실수가 아니라면, 즉 비트벡터가 Inf나 NaN을 의미한다면 각각 $\mathrm{f}_{\lceil i \rceil}$의 Inf와 NaN으로 변환되어야 한다. 따라서 $\mathrm{BV}_{\lceil i \rceil}$에서 Inf/NaN에 해당하는 MB로 변환한다.

- 변환하는 값이 실수라면, 즉 비트벡터가 의미하는 값이 Inf나 NaN이 아니라면 f_i에서 표현 가능한 실수는 정확히 동일한 값의 $\mathrm{f}_{\lceil i \rceil}$ 실수로 변환되어야 한다. 이는 변환된 $\mathrm{f}_{\lceil i \rceil}$ 실수가 f_i의 범위 내에 있으며 f_i에서도 정확하게 표현 가능한 값이라는 뜻이므로, $\mathrm{BV}_{\lceil i \rceil}$ 에서 EB와 OB는 모두 0이 되어야 한다. 따라서 BV_i 비트벡터의 MB앞에는 $|EB|$만큼의 0을, 뒤에는 $|OB|$만큼의 0을 붙여주어 $\mathrm{BV}_{\lceil i \rceil}$의 MB로 변환한다.

5.2.2 낮은 정밀도로 형변환

확장된 인코딩에서는 정해진 규칙에 따라 임의의 BV_i 비트벡터를 $\mathrm{BV}_{\lfloor i \rfloor}$ 비트벡터로 변환할 수 있다. 정밀도 변환을 거쳐도 값의 부호, 즉 SB는 동일하므로,

BV_i 비트벡터의 MB를 $\mathrm{BV}_{\lfloor i \rfloor}$의 MB로 변환한 뒤 BV_i 비트벡터의 SB를 앞에 붙여주면 $\mathrm{BV}_{\lfloor i \rfloor}$ 비트벡터가 된다.

5.2.1절에서 설명한 것과 같은 이유로 낮은 정밀도로 형변환하기 위한 규칙을 정해주어야 한다. BV_i 비트벡터의 MB를 $\mathrm{BV}_{\lfloor i \rfloor}$ 비트벡터의 MB로 변환하는 방법은 다음과 같다.

- 변환하는 값이 실수가 아니라면, 즉 비트벡터가 Inf나 NaN을 의미한다면 각각 $\mathrm{f}_{\lfloor i \rfloor}$의 Inf와 NaN으로 변환되어야 한다. 따라서 $\mathrm{BV}_{\lfloor i \rfloor}$에서 Inf/NaN에 해당하는 MB로 변환한다.

- 변환하는 값이 실수이고 $EB \neq 0$이라면, 즉 $\mathrm{f}_{\lfloor i \rfloor}$에서 표현 가능한 범위보다 큰 값이라면 $\mathrm{f}_{\lfloor i \rfloor}$의 Inf로 변환되어야 한다. 따라서 $\mathrm{BV}_{\lfloor i \rfloor}$에서 Inf에 해당하는 MB로 변환한다.

- 변환하는 값이 실수이고 EB=0, $OB \neq 0$이라면, 즉 $\mathrm{f}_{\lfloor i \rfloor}$에서 정확하게 표현할 수 없는 값이라면 $\mathrm{f}_{\lfloor i \rfloor}$의 *비슷한* 실수로 변환되어야 한다. 이 값은 $\mathrm{f}_{\lfloor i \rfloor}$로 반올림된 값이지만, 반올림 과정에서 실제로 값이 내려질지 올려질지는 알 수 없다.
  UF($(FB{+}{+}\ OB) \rightarrow$ bool) 를 정의하여 각 값에 임의로 반올림 방향을 부여하면 모든 경우의 수를 SMT solver에서 검증할 수 있다. 이때 반올림 방향이 올림이라면, 올림한 값은 내림한 값에 $\mathrm{f}_{\lfloor i \rfloor}$의 최소치 (epsilon) 을 더한 값이므로 올림한 $\mathrm{BV}_{\lfloor i \rfloor}$ 비트벡터의 MB는 내림한 $\mathrm{BV}_{\lfloor i \rfloor}$ 비트벡터의 $MB{+}\ 1$로 표현할 수 있다. 따라서 BV_i 비트벡터의 $FB{+}$ UF($FB{+}{+}\ OB$) ? 1 : 0로 변환한다.

- 변환하는 값이 실수이고 EB=0, OB=0이라면, 즉 $\mathrm{f}_{\lfloor i \rfloor}$에서 정확하게 표현 가능한 값이라면 $\mathrm{f}_{\lfloor i \rfloor}$의 동일한 실수로 변환되어야 한다. 이 값은 $\mathrm{f}_{\lfloor i \rfloor}$로 내림해도 정확히 같은 값이므로 규칙에 따라 BV_i 비트벡터의 FB는 $\mathrm{BV}_{\lfloor i \rfloor}$ 비트벡터의 MB와 일치해야 한다. 따라서 BV_i 비트벡터의 FB로 변환한다.

5.3 상수의 값 반영

$P_{(src,tgt)}$에 정밀도 변환이 존재한다면, 모든 f_i 상수는 $P_{(src,tgt)}$에 직접 등장하지 않은 $\mathrm{f}_{\lceil i \rceil}$ 상수로 변환될 수 있으며, 일부 $\mathrm{f}_{\lceil i \rceil}$ 상수는 $P_{(src,tgt)}$에 직접 등장하지 않은 f_i 상수로도 변환될 수 있다. 따라서 4.3절에서 설명한 것과 같은 이유로 서로 다른 정밀도로 표현된 상수 간의 관계도 검증 과정에 제대로 반영되도록 해야 한다.

$P_{(src,tgt)}$에 등장하는 부동소수점 타입 중 가장 높은 정밀도를 가지는 타입을 f_H, 가장 낮은 정밀도를 가지는 타입을 f_L라 가정한다. 먼저 $P_{(src,tgt)}$에 등장하는 모든 정밀도의 모든 상수를 f_H 상수로 형변환하여, $P_{(src,tgt)}$에 존재하는 f_H 상수로 간주한다. f_H 에서 $\mathrm{f}_{\lceil L \rceil}$까지의 부동소수점 타입 f_i에 대해, IEEE-754 형변환 규칙에 따라 모든 f_i 상수를 $\mathrm{f}_{\lfloor i \rfloor}$로 형변환하여, $P_{(src,tgt)}$에 존재하는 $\mathrm{f}_{\lfloor i \rfloor}$ 상수로 간주한다. 그 뒤 f_L에서 f_H까지 4.3.4 - 4.3.5절과 같은 과정을 거치되, $\mathrm{f}_{\lceil i \rceil}$ 상수를 인코딩한 비트벡터의 FB는 f_i로 반올림한 상수를 인코딩한 비트벡터의 MB와 동일하도록, 그리고 $\mathrm{BV}_{\lceil i \rceil}$에 대해 UF($FB$++ OB)는 실제 반올림 방향과 동일하도록 검증에 사용할 precondition을 넣어준다.

5.4 비트벡터의 너비 정하기

프로그램에 부동소수점의 정밀도 변환이 존재한다면, 각 변수에 들어간 값이 다음과 같이 극단적인 경우를 생각해볼 수 있다.

- $P_{(src,tgt)}$에 등장하는 모든 f_i 변수에 들어간 서로 다른 실수가 $\mathrm{f}_{\lfloor i \rfloor}$으로 반올림하면 동일한 실수일 수 있다.

- $P_{(src,tgt)}$에 등장하는 모든 f_i 변수에 들어간 서로 다른 실수가 $\mathrm{f}_{\lfloor i \rfloor}$으로 반올림하면 Inf일 수 있다.

만약 $|OB|$가 충분히 크지 않다면 서로 다른 BV_i 비트벡터 중 일부는 반드시 FB가 달라지게 되므로 첫 번째 경우를 정확하게 검증할 수 없다. 한편 $|EB|$가

충분히 크지 않다면 $\mathrm{f}_{\lfloor i \rfloor}$의 범위를 넘어가는 값의 가짓수가 부족하므로 두 번째 경우를 정확하게 검증할 수 없다. 4.4절에서 설명했듯 $|EB|$, $|OB|$가 너무 커져도 검증에 악영향을 미치므로, 적절한 $|EB|$, $|OB|$를 정하는 것이 중요하다.

$\#\mathrm{f}_i = k$라고 했을 때, $|EB|$와 $|OB|$는 다음과 같이 계산한다.

$$|EB| = \max(1, \lceil \log_2(k + 2^{|FB|+|OB|}) \rceil - (|FB| + |OB|)) \tag{5.1}$$

$$|OB| = \lceil \log_2(k+1) \rceil \tag{5.2}$$

$|EB|$ 는 최소 $\max(1, \lceil \log_2(k+2^{|FB|+|OB|}) \rceil - (|FB|+|OB|))$ 비트여야 한다. $P_{(src,tgt)}$에 등장하는 모든 f_i 변수가 $\mathrm{f}_{\lfloor i \rfloor}$로 표현 불가능한 범위의 값인 경우를 가정하면 $k \leq (2^{|EB|} - 1) \times 2^{|FB|+|OB|}$ 여야 하는 한편, $\mathrm{f}_{\lfloor i \rfloor}$로 표현 불가능한 범위의 값을 표현할 수 있으려면 $|EB|$가 최소한 1은 되어야 하기 때문이다. 하지만 다음에 설명할 $|OB|$가 최소 $\lceil \log_2(k+1) \rceil$ 이므로 이 식의 계산 결과는 $|FB|$나 $|OB|$가 지배 (dominate) 한다. 따라서 실질적으로 $|EB|$는 항상 1이 된다.

$|OB|$는 최소 $\lceil \log_2(N+1) \rceil$ 비트여야 한다. $P_{(src,tgt)}$에 등장하는 모든 f_i 변수가 $\mathrm{f}_{\lfloor i \rfloor}$로 정확하게 표현할 수 없지만 반올림한 결과가 동일한 경우를 가정하면, SB, EB, FB가 동일하기 때문에 OB로 표현 가능한 값이 최소 $N+1$ 개여야 각 변수에 서로 다른 값을 할당해줄 수 있기 때문이다.

제 6 장 최적화 검증 및 성능 비교

이번 장에서는 제안한 비트벡터 인코딩을 이용해 실제 머신러닝 컴파일러의 최적화를 검증한 결과를 소개한다. 머신러닝 컴파일러 프레임워크 MLIR에서 사용되는 최적화 패스 12개를 부동소수점 연산이 포함된 테스트 49개를 이용해 검증했으며, MLIR 최적화 검증 툴인 MLIR-TV를 일부 수정해 비트벡터 인코딩과 FPA 인코딩으로 최적화를 각각 검증하였다. 모든 테스트는 MLIR에서 자체적으로 최적화 검증을 위해 작성한 실제 사용되던 코드이다. FPA 인코딩에서 검증에 성공한 테스트를 비트벡터 인코딩으로 검증하면 항상 더 빠른 시간 안에 검증할 수 있었으며, FPA 인코딩을 사용했을 때는 연산 시간을 과도하게 소모해 SMT solver에서 timeout이 뜨던 테스트도 비트벡터 인코딩을 사용하자 합리적인 시간 내에 검증에 성공한 경우도 있었다.

한편 검증 과정에서 잘못 구현된 MLIR 최적화를 발견하였고, 이를 제보해 현재는 해당 최적화가 제거되었다.

6.1 실험 결과 비교

MLIR-TV에서 부동소수점 프로그램 검증에 비트벡터 인코딩을 사용한 경우와 FPA 인코딩을 사용한 결과를 비교하였다.

표 6.1 에서 BV는 비트벡터 인코딩, FPA는 FPA 인코딩을 사용해 검증한 결과를 나타낸다. #Tests는 검증에 성공한 테스트 / 전체 테스트 개수를 의미하며, 적색으로 표기한 부분은 테스트 중 하나 이상이 timeout으로 인해 검증에 실패한 경우를 나타낸다. Time은 해당 최적화 패스를 검증하기 위한 테스트를 전부 통과하기까지 걸린 시간을 의미하며, 적색으로 표기한 부분은 검증에 걸린 시간이 상대 인코딩에 비해 2배 이상 차이가 발생한 경우를 나타낸다.

대부분의 최적화 패스는 비트벡터 인코딩을 사용했을 때 더 빠른 시간 안에

	BV		FPA	
검증한 최적화 패스	#Tests	Time	#Tests	Time
conv2d-to-img2col	1/1	6.51s	1/1	128.36s
convert-elementwise-to-linalg	2/2	0.05s	2/2	0.09s
fold-memref-subview-op	4/4	0.87s	4/4	0.94s
fold-tensor-extract-op	3/3	0.09s	3/3	0.11s
fusion-tensor	12/12	7.22s	9/12	121.36s
linalg-bufferize	4/4	6.57s	4/4	147.51s
linalg-canonicalize	2/2	0.89s	2/2	25.81s
linalg-fold-unit-extent-dims	1/1	0.05s	1/1	3.77s
linalg-generalize-named-ops	2/2	0.78s	2/2	0.35s
tensor-constant-bufferize	2/2	0.07s	2/2	0.09s
tosa-make-broadcastable	1/1	0.07s	1/1	0.07s
tosa-to-linalg	15/15	7.15s	10/15	307.43s

표 6.1 비트벡터 인코딩과 FPA 인코딩을 이용해 최적화를 검증한 결과

검증이 끝났으며, FPA 인코딩을 사용했을 때 검증에 성공했지만 비트벡터 인코딩을 사용하자 실패한 경우는 없었다. 오히려 FPA 인코딩을 사용하면 SMT solver timeout으로 인해 검증에 실패하는 경우도 발생하였다. 특히 테스트에서 부동소수점 텐서가 등장하면 두 인코딩 간 검증 소요 시간 차이가 두드러졌는데, 이는 텐서가 부동소수점 변수를 수백 개 단위로 다루기 때문에 두 인코딩 간의 성능 차이가 극대화되기 때문이다.

최적화 패스에 따라 차이가 크지만 timeout으로 인해 비교가 어려운 테스트를 제외하면 비트벡터 인코딩이 같은 최적화 패스를 최대 75.4배까지 더 빠르게 검증했으며, 한 가지 최적화를 제외하면 FPA 인코딩보다 검증이 늦게 끝나는 경우는 없었다. 전체 검증에 걸린 시간으로 비교하면 동일한 최적화들을 비트벡터 인코딩이 FPA 인코딩보다 약 14배 더 빠르게 검증했다.

다만 linalg-generalize-named-ops 패스는 오히려 FPA 인코딩에서 더 빠르게 검증이 끝났는데, 이는 테스트들이 모두 값에 대한 연산 없이 단순히 복사한 결과가 동일한지 검증하는 테스트였기 때문에 비트벡터 인코딩의 장점을 살리기 힘든 반면 FPA 인코딩에 포함된 최적화를 적용하기 용이했기 때문으로 보인다.

6.2 탐지한 잘못된 최적화

일부 테스트에서 최적화 전후 프로그램이 서로 다른 값을 반환해 검증에 실패하는 경우가 발생했는데, 분석 결과 MLIR에서 사용 중이던 일부 최적화에서 덧셈의 항등원으로 -0.0이 아닌 0.0을 사용하고 있는 경우를 발견하였다[6]. 이 경우 입력값으로 -0.0이 들어가면 올바르지 않은 값이 반환되므로 틀린 최적화가 되며, 조사 결과를 제보해 현재는 해당 최적화가 제거되었다.

동일한 테스트를 FPA 인코딩을 사용해 검증했을 때는 해당 오류를 탐지하지 못했는데, 연산 시간이 과도하게 소요되어 모두 timeout이 발생했기 때문이다. 이는 비트벡터 인코딩이 훨씬 적은 연산량으로도 FPA 인코딩에 준하는 검증 능력을 보일 뿐만 아니라 경우에 따라서는 더욱 효과적으로 검증이 가능함을 증명하는 예시라고 볼 수 있다.

참고문헌

[1] C. Lattner et al., "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Seoul, Korea (South), 2021, pp. 2-14.

[2] S. Bang, S. Nam, I. Chun, H. Y. Jhoo and J. Lee, "SMT-Based Translation Validation for Machine Learning Compiler," in *Computer Aided Verification: 34th International Conference, CAV 2022, Proceedings, Part II*, Haifa, Israel, Aug 2022. pp. 386-407.

[3] Dialects, https://mlir.llvm.org/docs/Dialects/

[4] 'tensor' Dialect, https://mlir.llvm.org/docs/Dialects/TensorOps/

[5] Tensor Operator Set Architecture (TOSA) Dialect, https://mlir.llvm.org/docs/Dialects/TOSA/

[6] Canonicalization of 'x + (+0.0)' in tosa, https://discourse.llvm.org/t/canonicalization-of-x-0-0-in-tosa/4760

[7] "IEEE Standard for Floating-Point Arithmetic," in *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 22 July 2019, pp. 1-84.

[8] Rümmer, Philipp and Thomas Wahl., "An SMT-LIB Theory of Binary Floating-Point Arithmetic," 2010.

Abstract

An Efficient SMT Encoding for the Validation of Floating-Point Compiler Optimizations

Seunghyeon Nam
School of Computer Science Engineering
Collage of Engineering
The Graduate School
Seoul National University

This paper introduces a novel method to encode the floating-point (FP) value as SMT bit-vector and the operations between them as SMT formulas, which allows efficient yet accurate verification of FP compiler optimizations, and checks its effectiveness by experiments. It is essentially an overapproximation scheme for FP operations, which is less complex and more SMT-friendly compared to the IEEE-754 standard but maintains enough representative power for verification. When verifying the optimization passes from MLIR compiler framework using both abstract and concrete FP encoding, the abstract encoding could verify the same optimizations more quickly, and even allowed more passes to be verified than the concrete encoding due to less timeouts. The new encoding played a crucial role in detecting and removing some faulty MLIR optimizations.

Keywords: Program Verification, SMT solver, Floating-point

Student Number: 2021-22699