Google

Poison Semantics for MLIR

Jakub Kuderski <kubak@google.com>
(@kuhar on Discourse & Phabricator & GitHub)

MLIR Open Meeting, 2022-11-03

Public RFC:
https://discourse.llvm.org/t/rfc-poison-semantics-for-mlir/66245

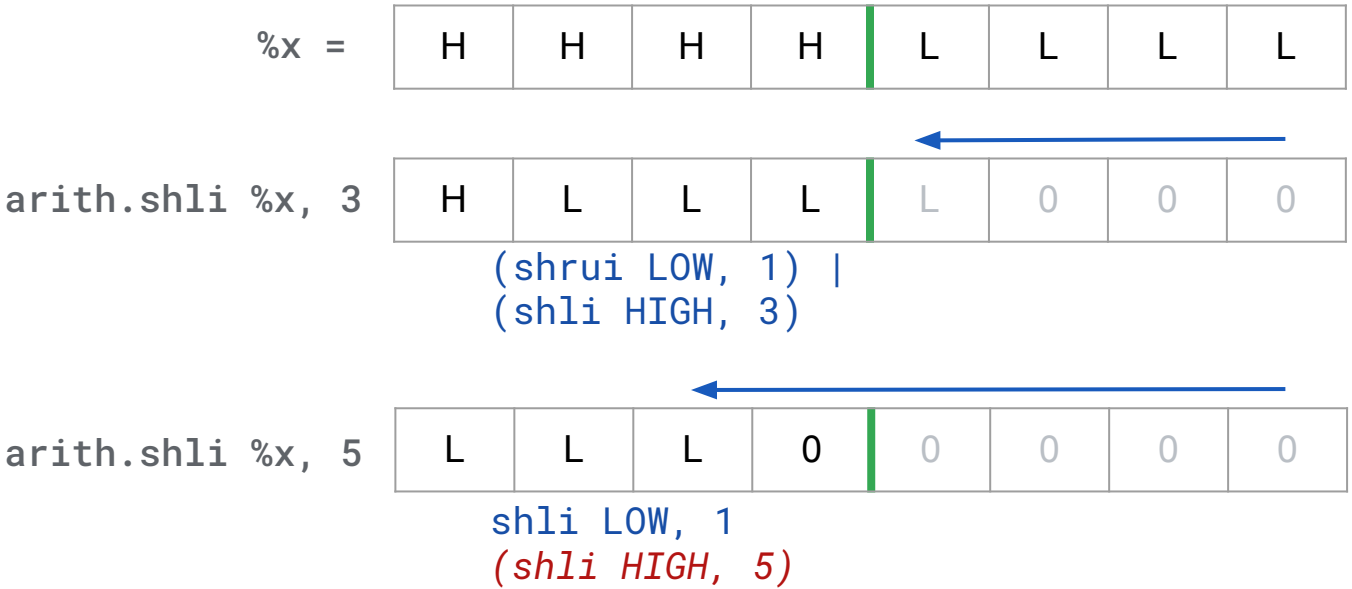
Agenda

- Motivating Example
- Goals
- Undefined Behavior & Speculation
- Poison Semantics
- Implementation Strategy

Motivation

Example from Wide Integer Emulation

Implementing 8-bit arithmetic 'shift left' using 4-bit ops



Example from Wide Integer Emulation

Implementing 8-bit arithmetic shift left using 4-bit ops

```
%x = arith.shli  %a, %p : i4
%y = arith.shrui %a, %q : i4
%z = arith.select %cond, %x, %y : i4
```

- Is this safe knowing that either %p or %q is >= 4?
- Do we need to introduce 'runtime checks'?
  - Control flow?
  - Clamp shift values?
  - Use saturating shifts?

Undefined Behavior & Speculation

Examples: Defined and Undefined Behavior

```
arith.addi %x, %y : i4
```

Well-defined for all inputs

```
arith.divui %x, 0 : i4
```

Undefined behavior:

- Program can do nothing or anything
- We assume it does not happen
- Cannot always speculate

```
arith.shrui %x, 5 : i4
```

Not well-defined for all inputs:

- Also undefined behavior?
- Undefined result but safe?
- If safe, how to use it?
- Consistent results?

Examples: Continued

```
unsigned x0 = a >> b;
if (x0 != 0) {
  unsigned x1 = a >> b;
  unsigned r = y / x1;
  foo(r);
}
```

```
arith.shrui %x, 5 : i4
```

Not well-defined for all inputs:

- Also undefined behavior?
- Undefined result but safe?
- If safe, how to use it?
- Consistent results?

Goals

Goal 1: Define op semantics

- Differentiate between different classes of 'illegal' inputs
- We need precise language to talk about 'undefined' results
- Define how ops produce, propagate, and consume them
- Make op behavior consistent and intuitive

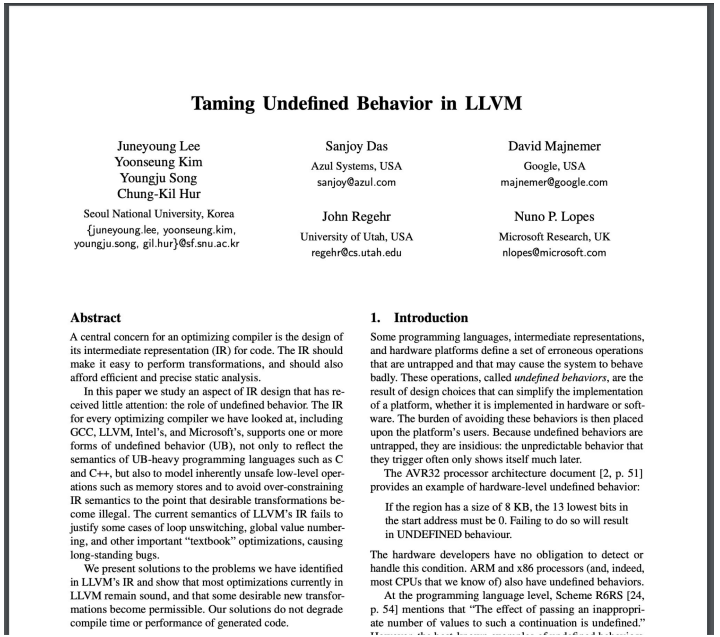
Goal 2: Provide building blocks for analyses

- Enable reasoning about illegal outputs and undefined results
- General enough to support open set of types and ops

Poison Semantics

Poison in LLVM

- Closed set of types and instructions, simple type system
- `poison` is an extra value, e.g.: `i8 := {0..255} ∪ poison`
- A 'virtual' value separate from the bitvector representation
- Instructions can:
  - **Produce** poison
  - **Propagate** poison
  - **Stop** poison propagation
  - Trigger immediate **Undefined Behavior**



Taming Undefined Behavior in LLVM

Juneyoung Lee
Yoonseung Kim
Youngju Song
Chung-Kil Hur
Seoul National University, Korea
{juneyoung.lee, yoonseung.kim, youngju.song, gil.hur}@sf.snu.ac.kr

Sanjoy Das
Azul Systems, USA
sanjoy@azul.com

David Majnemer
Google, USA
majnemer@google.com

John Regehr
University of Utah, USA
regehr@cs.utah.edu

Nuno P. Lopes
Microsoft Research, UK
nlopes@microsoft.com

Abstract

A central concern for an optimizing compiler is the design of its intermediate representation (IR) for code. The IR should make it easy to perform transformations, and should also afford efficient and precise static analysis.

In this paper we study an aspect of IR design that has received little attention: the role of undefined behavior. The IR for every optimizing compiler we have looked at, including GCC, LLVM, Intel's, and Microsoft's, supports one or more forms of undefined behavior (UB), not only to reflect the semantics of UB-heavy programming languages such as C and C++, but also to model inherently unsafe low-level operations such as memory stores and to avoid over-constraining IR semantics to the point that desirable transformations become illegal. The current semantics of LLVM's IR fails to justify some cases of loop unswitching, global value numbering, and other important "textbook" optimizations, causing long-standing bugs.

We present solutions to the problems we have identified in LLVM's IR and show that most optimizations currently in LLVM remain sound, and that some desirable new transformations become permissible. Our solutions do not degrade compile time or performance of generated code.

1. Introduction

Some programming languages, intermediate representations, and hardware platforms define a set of erroneous operations that are untrapped and that may cause the system to behave badly. These operations, called *undefined behaviors*, are the result of design choices that can simplify the implementation of a platform, whether it is implemented in hardware or software. The burden of avoiding these behaviors is then placed upon the platform's users. Because undefined behaviors are untrapped, they are insidious: the unpredictable behavior that they trigger often only shows itself much later.

The AVR32 processor architecture document [2, p. 51] provides an example of hardware-level undefined behavior:

> If the region has a size of 8 KB, the 13 lowest bits in the start address must be 0. Failing to do so will result in UNDEFINED behaviour.

The hardware developers have no obligation to detect or handle this condition. ARM and x86 processors (and, indeed, most CPUs that we know of) also have undefined behaviors.

At the programming language level, Scheme R6RS [24, p. 54] mentions that "The effect of passing an inappropriate number of values to such a continuation is undefined."

[Taming Undefined Behavior in LLVM, PLDI'17]

Poison in LLVM -- `poison` and `freeze`

- New constant: `poison` -- creates a poison value of a given type
- `freeze ty %x` -- stops poison propagation:
  - noop if `%x` is a well-defined value
  - produces a nondeterministic (arbitrary) value if `poison`
- Translation validation tool: https://alive2.llvm.org/ce/
  - **Alive2: Bounded Translation Validation for LLVM, PLDI'21**

Poison in LLVM -- Example

```
define i8 @src(i8 %arg0) {
  %x = ashr i8 %arg0, 9                ; poison
  %y = add i8 %x, %arg0                ; poison
  %z = select i1 false, i8 42, i8 %y   ; poison
  %r = freeze i8 %z                    ; nondet
  ret i8 %r
}

 ==>

define i8 @tgt(i8 %arg0) {
  %r = freeze i8 %arg0
  ret i8 %r
}
```

Poison semantics for MLIR -- Requirements

- Needs to support an open set of types and ops
- Opt-in:
  - By default, types and ops do **not** have poison semantics
- New dialect: 'ub'
  - New ops: `ub.poison`, `ub.freeze`, `ub.unreachable`
  - Ops available for types that opted in

Poison semantics for MLIR -- Scalar Case

Similar to LLVM, a value can either be well-defined or poison

- `ub.poison : ty` -- creates a value of type 'ty' in its poison state
- `ub.freeze %x : ty` -- noop for well-defined values, arbitrary value when `%x` is poison

A use of a `poison` value can result in one of the following, depending on the op:

1. Another `poison`, e.g., `arith.addi %x, poison : i32`
2. A well-defined non-`poison` value, e.g., `arith.select false, poison, 1 : i32`
3. Immediate UB, e.g., `arith.divui poison, 0: i32`

Poison semantics for MLIR -- Vector Case

Similar to LLVM, one bit of poison per vector element

- Enables scalarization/vectorization:

```
%x = arith.add %arg0, %arg0 : vector<2xi32>
%y = vector.extract %x [0] : vector<2xi32>
%z = vector.extract %x [1] : vector<2xi32>
 <==>
%a = vector.extract %arg0 [0] : vector<2xi32>
%b = vector.extract %arg0 [1] : vector<2xi32>
%y = arith.add %a, %a : i32
%z = arith.add %b, %b : i32
```

```
%v = ub.poison : vector<2xi32>
%c = arith.constant 0 : i32
%y = vector.insert %c into %v [0] : vector<2xi32>
%z = vector.extract %v [0] : vector<2xi32>
 ==>
%z = arith.constant 0 : i32
```

Poison semantics for MLIR -- Compound Types

- A hypothetical type: `pair.pair<ty1, ty2>`
- How to combine poison semantics of both element types?
- Some options:
  - Per-element poison semantics
  - Lowest common denominator of element types
  - Whole-value poison semantics

Implementation

Semantics

- Should be detailed enough to write an interpreter
- Caveat: must be consistent without over-relying on poison state
  - Virtual and does not exist
  - Valid to replace poison with a well-defined value

Implementation

- Used for analyses and transforms in the compiler
- Should allow for efficient sparse representation
  - E.g., for selected tensor elements
- Should stay close to the semantics but allow for over-approximation
  - E.g., shared poison state for scalable vector dimensions
- Two types of analyses: may-be-poison, may-not-be-poison

Implementation Strategy

- Bottom up:
  - Start with lowest-level dialects first: 'LLVM', 'arith', 'vector'
  - Gradually generalize the implementation
    - Allow for custom types/dialects to opt in
    - Add the 'ub' dialect
- Do not break existing dialects:
  - If none of the ops produce poison, there is nothing to update

Implementation Strategy -- When to Opt-in?

- Lowering must be a refinement
  - Can only remove undefinedness, but not make defined behavior undefined

- Tradeoffs:
  - Semantic mismatch leads to 'runtime checks'
  - Poison allows to exploit undefinedness in the name of performance
  - **Poison adds significant semantics complexity**

Implementation: Type and op interfaces

- Not 100% clear to me
- Type interface -- to what extent type supports poison semantics
- Op interface -- given a list of input poison states, what are the output poison states

A possible type interface:

1. `bool canBePoison()` – whether the type is poisonable and `ub.poison` can be used on this type
2. `bool canBeFrozen()` – whether the type is freezable and `ub.freeze` can be used on this type

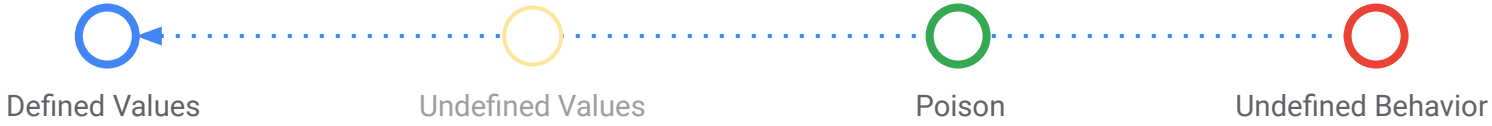
Thank You

Questions?

Bonus Slides

Refinement

When lowering, we can only make the program behavior more defined (or same)



Example from Wide Integer Emulation

Is this transform correct knowing some elements may be shifted by >= bitwidth?

```
%r  = arith.shrui %x, %y   : vector<2xi32>
%e0 = vector.extract %r[0] : vector<2xi32>
%e1 = vector.extract %r[1] : vector<2xi32>
 <==>
%x0 = vector.extract %x[0] : vector<2xi32>
%x1 = vector.extract %x[1] : vector<2xi32>
%y0 = vector.extract %y[0] : vector<2xi32>
%y1 = vector.extract %y[1] : vector<2xi32>
%e0 = arith.shrui %x0, %y0 : i32
%e1 = arith.shrui %x1, %y1 : i32
```