Write-and-Do-Mini-Assignment#1: SCeV in LLVM

Sagar Jain
CS17BTECH11034

September 2, 2019

Introduction

SCeV stands for **Scalar Evolution**. Scalar Evolution is essentially a technique to express how the value of a ***scalar*** variable changes (***evolves***) over iterations of a loop. Scalar evolution as a technique is immensely powerful, being able to express, co-relate(different variables) and simplify complex changes that occur through a loop. These properties have made SCeV a very useful analysis for optimisations in compilers.

SCeV in LLVM

The implementation of SCeV in LLVM is limited to those properties that can be used most readily by optimisers, so not everything that is theoretically possible using SCeV is used in LLVM.
Scalar Evolution in LLVM is actually implemented as an ***analysis pass*** which does not really lead to any transformation of the module but provides valuable information in the form of ***loop trip counter***, recurrence triplets for variables, etc. It can be used with opt in the following way:

```
opt -analyze -scalar-evolution <filename>
```

There are several loop transformations that use this information provided by scalar evolution to optimise the loop, for example: **Induction Variable Simplify** i.e. `indvars`, **Loop strength reduction** i.e. `loop-reduce`, etc.

Induction variables are broadly classified into two types:

1. Basic Induction Variable (BIV): Which increase or decrease by a constant on each iteration of the loop.
2. Generalized Induction Variable (GIV): More complex updates, may even depend on ther IVs.

The recurrence relation of any IV can be represented as a 3-tuple {a, b, c}, where a is the initial value, b categorises the type of operation and c is the value that the operand takes. Chaining of recurrences to create new recurrences is also allowed. The entire mathematical formalism of SCeV is illustrated in some detail at http://llvm.org/devmtg/2018-04/slides/Absar-ScalarEvolution.pdf

Observations on usage of SCeV in LLVM

The following are a few points to note before trying to fiddle with `-scalar-evolution`.

- Optimisations must to be run on IR files.
- If we use clang -emit-llvm ⟨filename⟩ -S, we get a .ll file but optimisations will not work on it since by default the attribute `optnone` is added to functions which do not allow optimisations to run on them.
- We can run clang with the following options to avoid the addition of `optnone attribute`. `clang -O0 -Xclang -disable-O0-optnone -emit-llvm ⟨filename⟩ -S`
- Trying to run scalar evolution on the output file of the above program also does not yield results, this is because most optimisation and analysis passes require the IR to be in pure SSA but the IR generated by us can even have loads and stores, to get around this we should first convert into SSA by using `opt -mem2reg ⟨filename⟩ -S`.
- Now, we can use `opt -analyze -scalar-evolution ⟨filename⟩` to get the desired output.

Basic Induction Variable

A typical loop counter would have a SCeV analysis similar to the following:

```
%inc = add nsw i32 %i.0, 1
--> {1,+,1}<nuw><%for.cond> U: [1,-2147483647) S: [1,-2147483647)
```

We can see that we have the recurrence tuple, signed and unsigned range and other information present for the instruction. Similarly, any instruction which is ***SCeVable*** has its SCeV values printed.