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

Sagar Jain
CS17BTECH11034

September 4, 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 Variables

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. Another Example:

```
%add = add nsw i32 %b.0, 2
--> {2,+,2}<nuw><nsw><%for.cond> U: [2,1003) S: [2,1003) Exits:  1002
```

In this example we get even more information in the form of exit value and more precise range.

Generalized Induction Variables

The example in the slides gives us the following code:

```
for ( int x = 0; x < n; x++)
p[x] = x*x*x + 2*x*x + 3*x + 7;
```

SCeV analysis in the above example happens according to the rewriting and folding rules for various arithmetic operations, we end up with the ***chained recurrence*** expressed as `{7,+,6,+10,+,6}`. This takes place in a bottom up fashion where we build the recurrence for complex expression by combining the recurrences for simpler BIVs and GIVs.

Example where SCeV can be used to simplify expressions:

```
void foo(int *a) {
for (int i = 0; i < 100; i++)
a[i] = (i+1)*(i+1) - i*i - 2*i; // equals 1
}
```

We know that the expression always yields the constant 1 in every iteration of the loop and we do not need to perform all the multiplication and addition everytime. SCeV arrives at the same conclusion when we get the final recurrence as:
`{1,+,3,+,2} - {0,+,3,+,2} = {1,+,0,+,0}`
Which can be written just as `(1)`, reducing lot of computation within the loop.

Loop Strength Reduction Using SCeV

The following gives an example of how SCeV can be used to reduce the strength of loops.

```
void foo(int *a, int n, int c, int k) {
for (int i = 3; i < n; i++) {
a[c*i] = c*i+k;
}
}
```

The recurrences come out to be:

```
scev(%i) = {3,+,1}
SCEV(%ci) = {(3*%c),+,%c}
scev(%ci_plus_k) = {(3*%c+%k),+,%c}
SCEV(%arrayidx) = {(12*%c +%a),+,(4*%c)}
```

LSR then collects all the recurrences which have chains in them. The next step is to identify all the common expressions in them and identify a way to compute all of them to use the least amount of resources i.e. both cycles and storage. In this process we also end up getting better alternative operations like addition of %c to the previous number rather than multiplying %c with i in every iteration.
These are just a few applications of SCeV, it is used in several other optimisations as well as mentioned above.