Project 2

Advanced Compilers 2024

TA: Juneyoung Park (`juneyoung.park@snu.ac.kr`)

May 13, 2024

Due: Jun 4, 2024

By using Static Single Assignment (SSA) Intermediate Representation (IR), the well-known Sparse Conditional Constant Propagation (SCCP) algorithm enjoys fast and powerful transform [1]. Many modern compilers, including LLVM[1], implement and use SCCP widely. In this project, we will implement simple SCCP analysis for integer constants as LLVM pass plugin.

1 Background

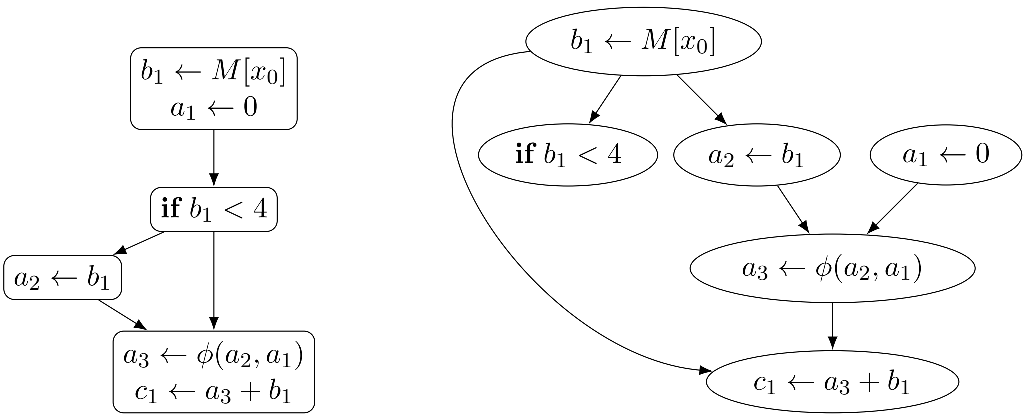
1.1 SSA

1.1.1 Dominating relationship

In SSA-formed language, each value of a variable are statically distinguishable. Thus, *Use*s of a value are correlated with a unique *Def* of a value. In other words, in SSA, all *Use*s of a variable are dominated by the variable's *Def*.

1.1.2 SSA graph

Due to the dominating relationships in SSA, *Def-Use* and *Use-Def* chains are condensed. Since, corresponding definition of a variable is unique to its *Use*s, one can build a graph based on variables' *Use-Def* chains called an SSA graph.



(a) CFG (b) SSA Graph

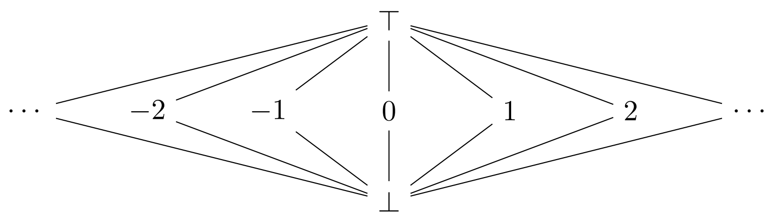
Figure 1: CFG and corresponding SSA graph.

[1] LLVM's implementation additionally uses the dominator tree to manage unreachable blocks.

2 Sparse Conditional Constant Propagation [1]

Just like traditional dataflow analyses, SCCP algorithm travels the Control Flow Graph (CFG). However, unlike traditional analyses, SCCP also dissemniates dataflow information through SSA graph edges. Thanks to the sparsity of SSA graph and dominating relationships inherent in SSA form, the SCCP can propagate the information more efficiently than before. Also, SCCP keeps evaluating branch conditions (conditionals) and executable paths with its best knowledge. Thus, SCCP can determine a variable's constancy, dependent on branches with constant conditions.

2.1 Lattice



$$
\begin{aligned}
\top &: \text{Undefined} \\
C := \{\dots, -1, 0, 1, \dots\} &: \text{Integer constant} \\
\bot &: \text{Not a constant}
\end{aligned}
$$

2.1.1 Meet[2]

$$
\begin{aligned}
x \sqcap \top =& x \\
x \sqcap \bot =& \bot \\
c_1 \sqcap c_2 =& c_1 \quad (c_1 = c_2) \\
c_1 \sqcap c_2 =& \bot \quad (c_1 \neq c_2)
\end{aligned}
$$

[2]Since '$\wedge$' and '$\vee$' can be confused with logical AND and OR, '$\sqcap$' and '$\sqcup$' are used to denote meet and join, respectively.

2.2 Algorithm[2, p. 101]

Algorithm 1 Sparse Conditional Constant Propagation

```
Input: CFG, SSA graph
CFGWorklist ← {NULL→ entry}
SSAWorklist ← ∅
while ¬CFGWorklist.Empty ∧¬SSAWorklist.Empty do
    x ←pop one from any list
    if x is from CFGWorklist then
        B ← x.Destination
        B.Executable ← true
        visit(φ)    (∀φ ∈ B)
        if B.IsFirstVisit then
            visit(I)    (∀I ∈ B)
        if ∃!B.Successor ∧ ¬B.Successor.Executable then
            CFGWorklist.append(B.Successor)
    else /* x is from SSAWorklist */
        if x is a φ then
            visit(x)
        else
            if Any of the incoming CFG edge to x is Executable then
                visit(x)
```

Algorithm 2 visit

```
Input: An instruction I
if I is a φ then
    Meet all the arguments from the executable edges.
else if I is a conditional branch then
    Evaluate the condition and append reachable edge to CFGWorklist.
else
    Update dataflow information using transfer function.

if Dataflow information of I had been changed then
    Append all outgoing SSA edges of I to SSAWorklist.
```

3 Code

3.1 `LatticeValue`

Base class for lattice value. This class uses CRTP.

3.2 `class SparseDataflowFramework`

Base class for analysis pass. In this project, our pass will be applied to function units.

3.3 `getId`

Returns unique id (name) of given `Value`. Note that some LLVM instructions may not have names (e.g., `br`, `ret`).

3.4 `ConstantValue`

Class for the lattice value of simple SCCP.

3.5 `CFGEdge`

Edge of CFG. `nullptr` is used to denote anonymous block outside the CFG. Also some relational operators are implemented due to meet ordering conditions of some container classes.

3.6 `SimpleSCCP`

Analysis pass implementing 'simple SCCP' for integer constants.

3.7 `SimpleSCCP::InstVisitor`

Instruction visitor used while implementing 'visit'. This struct inherits LLVM's instruction visitor.

3.8 `SimpleSCCPPrinter`

Printer pass of '`SimpleSCCP`'. If you wish to print the analysis result of `SimpleSCCP` to stream, use this helper pass.

Tips

- Please read LLVM Programmer's Manual and Doxygen References. Since, the doxygen references are generated on the latest version of LLVM, be **cautious** that there can be differences between the version we use (LLVM-17).
- One can find source code of LLVM at github. ('`llvm/lib/Transforms/Scalar/SCCP.cpp`', '`llvm/lib/Transforms/Utils/SCCPSolver.cpp`')
- Most of the containers' `operator[]` generates a default entry if there is no corresponding entry. Consider using `find` or other methods that do not automatically insert entries.
- `opt` prints out the IR through `stdout`. Informations and statistics are usually printed to `stderr`.
- One should clean out `build` directory before building other projects.

4 Input

Your analysis pass will get an IR, after '`mem2reg`' pass. `mem2reg` pass promotes variables reside in stack to virtual registers and prune cumbersome SSA structures. Please refer to the 'Commands' section below for the commands to generate input IR.

5 Objectives

1. Implement all three `TODO`s in '`SimpleSCCP.cpp`'.
   - Your implementation should not change the given IR.
   - Printing order can be nondeterministic by the data structure (ADT) being used.

6 Commands

- Build your plugin.

```
cmake -S project2 -G Ninja -B build
```

- Generate IR from the source.

```
clang -O0 -S -emit-llvm -Xclang -disable-O0-optnone -Xclang -disable-llvm-passes
-fno-discard-value-names test.c -o test.ll
```

- Run ‘mem2reg’ pass.

```
opt -S -passes='mem2reg'./test.ll -o input.ll
```

- Call your pass from opt.

```
opt -S -load-pass-plugin build/lib/libSimpleSCCP.so -passes='print<simple-sccp>'
./input.ll -disable-output 2> test.out
```

7 Submission

- Compress and submit your SimpleSCCP.cpp as PR2_<student_id>.zip (e.g., PR2_2024-12345.zip) at eTL.
- If you have modified any file other than SimpleSCCP.cpp or added new files, compress and submit entire project directory, including **CMakeLists.txt**, into a single zip file named after your student id (e.g., PR2_2024-12345.zip).

8 Example

```
%z.0 : { 5 }
%add1 : { 5 }
%add : { 3 }
%cmp : { 0 }
```

Figure 2: SimpleSCCP pass’ output example (test.out)

References

[1] M. N. Wegman and F. K. Zadeck, “Constant propagation with conditional branches,” *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 2, pp. 181–210, Apr. 1991, ISSN: 0164-0925. DOI: 10.1145/103135.103136. [Online]. Available: https://doi.org/10.1145/103135.103136.

[2] F. Rastello and F. Tichadou, *SSA-based Compiler Design*. Springer International Publishing, 2022, pp. 95–106, ISBN: 9783030805142. [Online]. Available: https://books.google.co.kr/books?id=zS54zgEACAAJ.