Program Analysis - Herbrand Equivalence

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Himanshu Rai
(111601032)

under the guidance of

Dr Jasine Babu

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

<u>CERTIFICATE</u>

*This is to certify that the work contained in this thesis entitled "**Program Analysis - Herbrand Equivalence**" is a bonafide work of **Himanshu Rai (Roll No. 111601032)**, carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my supervision and that it has not been submitted elsewhere for a degree.*

Dr Jasine Babu

Assistant/Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Contents

List of Figures iii

1 Introduction 1

1.1 Herbrand Equivalence 2

1.2 Organization of The Report 4

2 Review of Prior Works 5

3 Summary of Gulwani and Necula 7

3.1 Brief overview of the algorithm 8

3.2 Complexity of the algorithm 9

4 Summary of Saleena and Paleri 11

4.1 Notation 11

4.2 Value Expression 12

4.3 Algorithm 12

5 Summary of Babu, Krishnan and Paleri 13

5.1 Program Expressions 14

5.2 Congruence Relation 14

5.3 Transfer function 14

5.4 Non deterministic assignment 15

5.5 Dataflow analysis Framework 15

5.6 Herbrand Equivalence 15

6 Running a LLVM Pass **17**

6.1 Installing Clang 17

6.2 Building LLVM from source 17

6.3 Getting LLVM IR using Clang 18

6.4 Writing a pass 19

6.5 Running a pass 21

7 Implementation **23**

8 Conclusion and Future Work **27**

References **29**

List of Figures

1.1 Example of Herbrand Equivalence 3

1.2 Example of Herbrand Equivalence analysis at a confluence point 4

3.1 Computation of SED for flowgraph nodes of a program 8

6.1 Various stages of compilation using Clang 18

Chapter 1

Introduction

The basic job of a compiler is code translation from a high level language to a target assembly language. But, compilers also run multiple optimization pass in the intermediate stages of translation, so that the finally generated code performs better than just a normal translated code. There might be a one time overhead of running optimizations, but the performance gain visible over multiple executions of the code outweighs it.

Modern compilers performs a large number of optimizations like induction variable analysis, loop interchange, loop invariant code motion, loop unrolling, global value numbering, dead code optimizations, constant folding and propagation, common subexpression elimination etc. One common feature of most of these optimizations is detecting equivalent program subexpressions.

Checking equivalence of program subexpressions has been shown to be an undecidable problem, even when all the conditional statements are considered as non deterministic. So, in most of the cases compilers try to find some restricted form of expression equivalence. One such form of expression equivalence is called *Herbrand Equivalence* (see below). Detecting equivalence of program subexpressions can be used for variety of applications. Compilers can use these to perform several of the optimizations mentioned above like constant propagation, common subexpression elimination etc. Program verification tools can use these equivalences to discover loop invariants and to verify program assertions. This information is also important for discovering equivalent computations in different programs, which can be used by plagiarism detection tools and translation validation tools [1, 2], which compare a program with an optimized version in order to check correctness of the optimizer.

1.1 Herbrand Equivalence

A formal definition of Herbrand Equivalence is given in [3]. Informally, two expressions are Herbrand equivalent at a program point, if and only if they have syntactically the same value at that particular point, across all the execution paths from the start of the program which reaches that point. For the purpose of analysis, the operators themselves are treated as uninterprated functions with no semantic significance, only syntactic information is taken into consideration.

For Herbrand equivalence analysis, we consider the set of all possible expressions that can be formed using the constants, variables and operators used in the program. And for each program point, partition them such that two expressions are Herbrand equivalent at that point if and only if they belong to the same partition class for that point.

Figure 1.1 shows a simple example of Herbrand Equivalence analysis. All the expressions that belongs to the same set at a program point are Herbrand equivalent at that point.

- Initially all the expressions are in separate sets, ie. they are inequivalent to each other. In particular, note that $X + 2$ and $2 + X$ are inequivalent because the operators are being treated uninterprated with no semantic information of them, which means there is no knowledge of commutativity of $+$.

- After assignment $X = 1$, any occurrence of X in an expression can be replaced with 1. So, now all expressions with 1 in place of X and vice versa are equivalent.

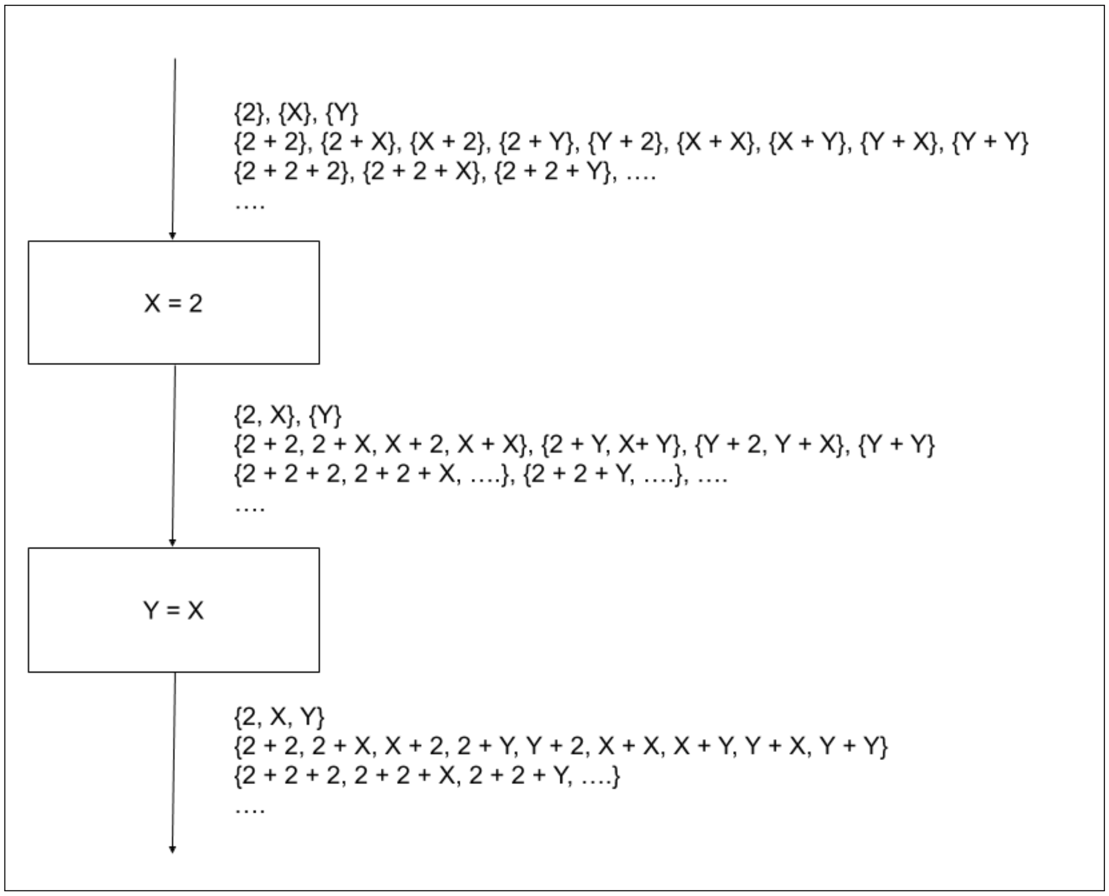
{2}, {X}, {Y}
{2 + 2}, {2 + X}, {X + 2}, {2 + Y}, {Y + 2}, {X + X}, {X + Y}, {Y + X}, {Y + Y}
{2 + 2 + 2}, {2 + 2 + X}, {2 + 2 + Y}, ...
....
X = 2
{2, X}, {Y}
{2 + 2, 2 + X, X + 2, X + X}, {2 + Y, X+ Y}, {Y + 2, Y + X}, {Y + Y}
{2 + 2 + 2, 2 + 2 + X, ...}, {2 + 2 + Y, ...}, ...
....
Y = X
{2, X, Y}
{2 + 2, 2 + X, X + 2, 2 + Y, Y + 2, X + X, X + Y, Y + X, Y + Y}
{2 + 2 + 2, 2 + 2 + X, 2 + 2 + Y, ...}
....

Fig. 1.1 Example of Herbrand Equivalence

- After assignment $Y = X$, any occurrence of Y in an expression can be replaced with X. Because X and 1 are equivalent, it means now 1, X, and Y are all equivalent to each other. And two expressions are equivalent if one can be obtained from the other by replacing one of these with any of the other two. For this example, it means that two expressions of the same length are equivalent.

Figure 1.2 shows what happens at a *confluence point* - a point where multiple paths meet. Two expressions are Herbrand equivalent only if they are Herbrand equivalent at all the predecessor points.

- In the left branch 2, Y, Z are equivalent and so are expressions obtained by replacement of any of these with any other.

- The case with the right branch is similar, except X and Y are interchanged.

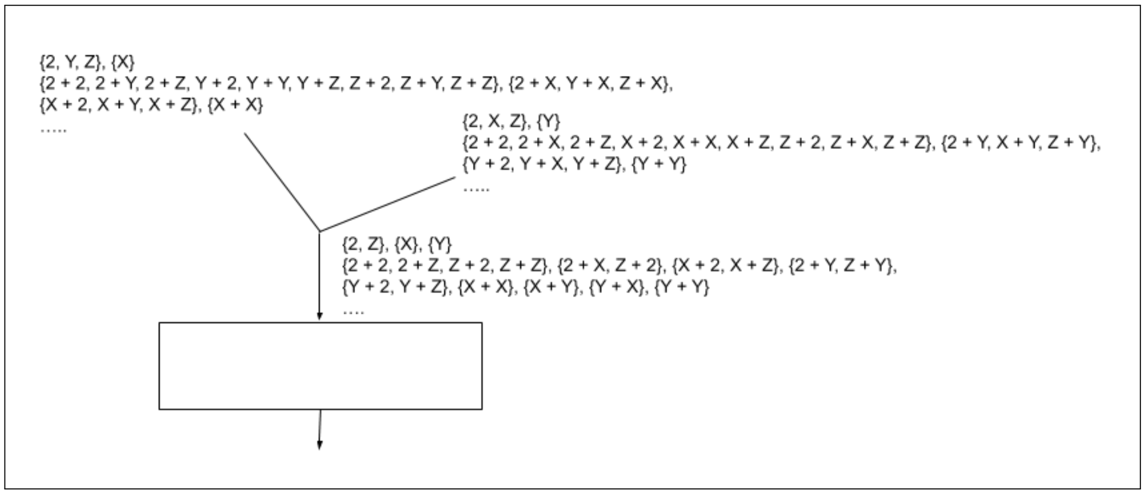
{2, Y, Z}, {X}
{2 + 2, 2 + Y, 2 + Z, Y + 2, Y + Y, Y + Z, Z + 2, Z + Y, Z + Z}, {2 + X, Y + X, Z + X},
{X + 2, X + Y, X + Z}, {X + X}
.....
{2, X, Z}, {Y}
{2 + 2, 2 + X, 2 + Z, X + 2, X + X, X + Z, Z + 2, Z + X, Z + Z}, {2 + Y, X + Y, Z + Y},
{Y + 2, Y + X, Y + Z}, {Y + Y}
.....
{2, Z}, {X}, {Y}
{2 + 2, 2 + Z, Z + 2, Z + Z}, {2 + X, Z + 2}, {X + 2, X + Z}, {2 + Y, Z + Y},
{Y + 2, Y + Z}, {X + X}, {X + Y}, {Y + X}, {Y + Y}
....

Fig. 1.2 Example of Herbrand Equivalence analysis at a confluence point

- At the confluence point, only 2 and Z are equivalent because they were equivalent at both the predecessors points. X was equivalent to 2 and Z at the right predecessor but not the left one and Y was equivalent to 2 and Z at the left predecessor but not the right. As before, expressions obtained by replacing 2 with Z and vice versa are equivalent.

1.2 Organization of The Report

Chapter 2 gives a brief overview of previous works related to the Herbrand Equivalence. Then a brief summary of the papers [4, 5, 6] is presented. Chapter 6 provides a tutorial on writing an LLVM optimzation pass. Finally, chapter 7 gives a pseudocode for Herbrand Equivalence analysis of a program, which is based on the algorithm mentioned in [6].

Chapter 2

Review of Prior Works

Existing algorithms for calculating Herbrand Equivalence are either exponential or are imprecise. The precise algorithms are based on an early algorithm by Kildall [7], which discovers equivalences by performing an abstract interpretation over the lattice of Herbrand equivalences. Kildall algorithms is precise in the sense it finds all the Herbrand equivalences but is exponential in time. The partition refinement algorithm of Alpern, Wegman and Zadek (AWZ) [8] is efficient but is much imprecise compared to Kildall's. AWZ algorithm represent the values of variables after a join using a fresh selection function ϕ_i, similar to functions in the static single assignment form and treats ϕ_i as uninterpreted functions. It is incomplete in the sense it treats all ϕ_i as uninterpreted. In an attempt to remedy this problem, Ruthing, Knoop and Steffen proposed a polynomial-time algorithm (RKS) [9] that alternately applies the AWZ algorithm and some rewrite rules for normalization of terms involving ϕ functions, until the congruence classes reach a fixed point. Their algorithm discovers more equivalences than the AWZ algorithm, but remains incomplete.

Gulwani and Necula [4] gave algorithm to find the Herbrand Equivalence classes restricted to program expressions. There algorithm is linear in parameter s, where s is the maximum times $+$ occurs in a program expression. Clearly s can take a maximum value of n, which is the program size, so the algorithm in all is polynomial in n. Later, Saleena and Paleri [5] showed that Gulwani's algorithm losses some information as it removes a equivalence class if it does not contain a variable or a constant. The global value numbering (GVN) algorithm proposed by them was able to detect more redundencies compared to that by Gulwani and Necula.

One problem is that most of these alogrithms were based on fix point computations but the classical definition of Herbrand equivalence is not a fix point based definition making it difficult to prove their precision or completeness. Babu, Krishnan and Paleri [6] developed a lattice theoretic fix-point formulation of Herbrand Equivalence on the lattice defined over the set of all terms constructible from variables, constants and operators of a program. They showed this definition is equivalent to the classical meet over all path characterization over the set of all possible expressions. The algorithm proposed by them is able to detect all the equivalences as by that of Saleena and Paleri.

So, to sum up Kildall's algorithm finds all the equivalent classes but is exponential. The algorithms by Saleena and Paleri; Babu, Krishnan and Paleri are polynomial and efficient among other imprecise algorithms. They are able to find all equivalence classes restricted to program expressions (all expressions with length atmost 2), which is precisely what is practically useful.

Chapter 3

Summary of Gulwani and Necula

Gulwani showed that there is a family of acyclic programs for which the set of all Herbrand equivalences requires requires an exponential sized (with respect to the size of the program) value graph representation - the data structure used by Kildall in his algorithm. He also showed that Herbrand Equivalences among program sub expressions can always be represented using linear sized value graph. This explains the reason for exponential complexity of Kildall's algorithm which cannot be improved to polynomial and imprecise nature of existing polynomial time algorithms.

So contrasting to Kildall's algorithm, which finds *all the Herbrand Equivalent classes* corresponding to constants, variables and operators occurring in the program, Gulwani's algorithm discovers *equivalences among program subexpressions* (expressions that can occur syntactically in a program), in linear time with respect to parameter s, the maximum size of an expression in terms of number of operators used. For global value numbering, s can be safely taken to be N, the size of the program and hence the algorithm is linear in the program size.

Also, he proved that the lattice of sets of Herbrand equivalences has finite height k, which is the number of program variables. So, an abstract interpretation over the lattice of Herbrand equivalences will terminate in at most k iterations even for cyclic programs.

3.1 Brief overview of the algorithm

The program expressions considered can be represented as

$$e ::= x \mid c \mid F(e_1, e_2)$$

Here, c and x are constants and variables occurring in the program.

The data structure used is called *Strong Equivalence DAG (SED).* Each node is of the form $< V, t >$ where V is a set of program variables and t is either $\perp$ or c for leaf nodes and $F(n_1, n_2)$ where n_1 and n_2 are SED nodes for non leaf nodes (also indicating that the node has two ordered successors). $\perp$ means that the variables in the node have undefined values.

There is a SED associated with each program point and the algorithm starts with the following initial SED

$$G_0 = \{< x, \perp > \mid x \text{ is a program variable}\}$$

Two functions *Join*(G_1, G_2, s') and *Assignment*$(G_1, x := e)$ are used to compute SEDs for other points in the flow graph node corresponding to the program, as shown in figure 3.1. s' in the argument of *Join* is a positive integer, and it returns equivalences between expressions of size atmost s'.

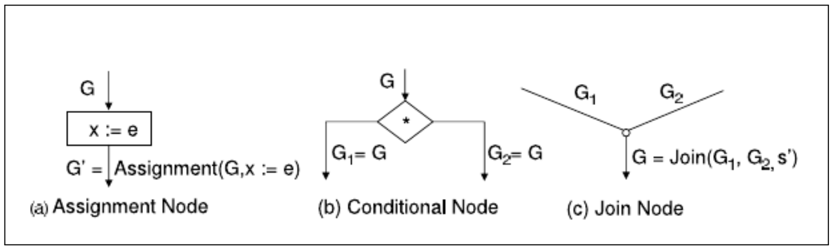

Fig. 3.1 Computation of SED for flowgraph nodes of a program

For detailed implementation of *Join* and *Assignment* and correctness proof of the algorithm, see [4].

3.2 Complexity of the algorithm

The complexity of the algorithm is $O(k^3 * N * j)$, where k is the total number of program variables, N is the size of the program and j is the number of join operations in the program. k and j are bounded by N, making the whole algorithm polynomial in N.

Chapter 4

Summary of Saleena and Paleri

Saleena and Paleri gave an algorithm for *global value numbering (GVN)*. GVN works by assigning a value number to variables and expressions. The same value number is assigned to those variables and expressions which are provably equivalent. A notable difference between Herbrand equivalence and GVN is that in Herbrand equivalence we talk about equivalences at a particular program point but in GVN are concerned with equivalence between expressions at two different program points.

The data structure used in the algorithm is called *value expression*. An expression with value numbers as operands, is called value expression. Two expressions are equivalent if they have same value expression. So, a value expression can be used to represent a set of equivalent program expressions.

4.1 Notation

Input is a flow graph atmost one assignment statement in each node which has one of the following forms

$$x \ ::= \ e$$

$$e \ ::= \ x \mid c \mid x \, op \, x$$

The flow graph also has two additional empty $ENTRY$ and $EXIT$ nodes. For a node n, IN_n and OUT_n denotes the input and output program points of the node.

Expression pool at a program point, is a partition of expressions at that point, in which equivalent expression belongs to the same partition. Each class will have a value number which we will consider as its first element. For a node n, EIN_n and $EOUT_n$ denotes the expression pools at input and output program points of the node.

4.2 Value Expression

The value expression corresponding to an expression is obtained by replacing actual operands with their corresponding value numbers. Example - For the expression-pool $\{\{v_1, a, x\}, \{v_2, b, y\}\}$ and statement $z ::= x + y$, the value-expression for $x + y$ will be $v_1 + v_2$. Instead of $x + y$, its value-expression is included in the expression-pool, with a new value number ie. the new expression-pool would be $\{\{v_1, a, x\}, \{v_2, b, y\}, \{v_3, v_1 + v_2, z\}\}$.

The value expression v_1+v_2 represents not just $x+y$ but the set of equivalent expressions $\{a + b, x + b, a + y, x + y\}$. Its presence indicates that an expression from this set is already computed and this information is enough for detection of redundant computations. Also, a single binary value expression can represent equivalence among any numbre of expressions of any length. Example - $v_1 + v_3$ represents, $a + z$, $x + z$, $a + (a + b)$, $a + (x + b)$ and so on.

4.3 Algorithm

Similar to Gulwani's algorithm, the algorithm consists of two main functions - a transfer function for changes in expression pool across assignment statements and a confluence function to find the expression pool at points were two branches meet. The algorithm starts with $EOUT_{ENTRY} = \phi$, and uses transfer and confluence functions to calculate expression pools at other points. This process is repeated till there is any change in the equivalence information. For detailed implementation refer to [5].

Chapter 5

Summary of Babu, Krishnan and Paleri

One of the problems with other former approaches to Herbrand equivalence is that most of the alogrithms were based on fix point computations. But the classical definition of Herbrand equivalence is not a fix point based definition making it difficult to prove their precision or completeness. Babu, Krishnan and Paleri [6] gave a new lattice theoretic formulation of Herbrand equivalences and proved its equivalence to the classical version.

The paper defines a congruence relation on the set of all possible expressions and shows that the set of all congruences for a complete lattice. Then for a given dataflow framework with n program points, a continuous composite transfer function is defined over the n-fold product of the above lattice such that the maximum fix point of the function yields the set of Herbrand equivalence classes at various program points. Finally, equivalence of this approach to the classical meet over all path definition of Herbrand Equivalence is established.

Below is a brief summary of the developments in the paper, for more detailed approach and proofs and for equivalence to MOP characterization refer to [6].

5.1 Program Expressions

Let C and X be the set of constants and variables occurring in the program respectively. The program expressions (terms) can be described as

$$t \ ::= \ c \mid x \mid t_1 + t_2$$

where $c \in C$ and $x \in X$.

5.2 Congruence Relation

Let T be the set of all program terms. A partition P of terms in T is said to be a congruence (of terms) if

- For t, t', s, $s' \in T$, $t' \cong t$ and $s' \cong s$ iff $t' + s' \cong t + s$.
- For $c \in C$, $t \in T$, if $t \cong c$ then either $t = c$ or $t \in X$.

Let $G(T)$ be the set of all congruences over T. We say $P_1 \preceq P_2$ for $P_1, P_2 \in G(T)$, if $\forall A_1 \in P_1, \exists A_2 \in P_2$ such that $A_1 \subseteq A_2$. We define *confluence* operation as

$$P_1 \wedge P_2 \ = \ \{A_i \cap B_j \mid A_i \in P_1 \text{ and } B_j \in P_2\}$$

Now, we extend $G(T)$ to $\overline{G(T)}$ by introducing abstract congruence $\top$ satisfying $P \wedge \top = \top, \forall P \in \overline{G(T)}$. Also, we denote the congruence in which every element is in a separate class as $\bot$.

$(\overline{G(T)}, \preceq, \bot, \top)$ forms a complete lattice, with $\wedge$ as meet operator.

5.3 Transfer function

An assignment $y := \beta$ transforms a congruence P to another congruence P'. This can be described in the form of transfer function $f_{y=\beta} : G(T) \rightarrow G(T)$, given by

- $B_i = \{t \in T \mid t[y \leftarrow \beta] \in A_i\},$ for each $A_i \in P$
- $f_{y=\beta}(P) = \{B_i \mid B_i \neq \phi\}$

We extend this definition to form extended transfer function, $\overline{f}_{y=\beta} : \overline{G(T)} \to \overline{G(T)}$ by defining $\overline{f}_{y=\beta}(\top) = \top$, otherwise $\overline{f}_{y=\beta}(P) = f_{y=\beta}(P)$. The extended transfer function is distributive, monotonic and continuous.

5.4 Non deterministic assignment

An assignment $y := *$ transforms a congruence P to another congruence P'. This can be described in the form of another transfer function $f_{y=*} : G(T) \to G(T)$, given by: for every t, $t' \in T$, $t \cong_{f(P)} t'$, (here $f(P) = f_{y=*}(P)$ for simplicity) iff

- $t \cong_P t'$
- $\forall \beta \in (T \setminus T(y)),\ t[y \leftarrow \beta] \cong_p t'[y \leftarrow \beta]$

As before we extend this transfer function to $\overline{f}_{y=*} : \overline{G(T)} \to \overline{G(T)}$ by defining $\overline{f}_{y=*}(\top) = \top$, otherwise $\overline{f}_{y=*}(P) = f_{y=*}(P)$. The function $\overline{f}_{y=*}$ is also continuous.

5.5 Dataflow analysis Framework

A dataflow framework over T is $D = (G, F)$ where $G(V, E)$ is the control flow graph associated with the program and F is a collection of transfer function associated with program points.

5.6 Herbrand Equivalence

The Herbrand Congruence function $H_D : V(G) \to \overline{G(T)}$ gives the Herbrand Congruence associated with each program point and is defined to be the maximum fix point of the *continuous composite transfer function* $f_D : \overline{G(T)}^n \to \overline{G(T)}^n$, where $\overline{G(T)}^n$ is the product lattice, f_D is a function satisfying $\pi_k \circ f_D = f_k$. Here π_k is the projection map and $f_k : \overline{G(T)}^n \to \overline{G(T)}$ is defined as follows

- If k = 1, the entry point of the program $f_k = \perp$.
- If k is a function point with $Pred(k) = \{j\},$ then $f_k = h_k \circ \pi_j$ where h_k is the extended transfer function corresponding to function point k.
- If k is a confluence point with $Pred(k) = i, j,$ then $f_k = \pi_{i,j},$ where $\pi_{i,j} : \overline{G(T)}^n \to \overline{G(T)}$ is given by $\pi_{i,j}(P_1, \ \ldots, \ P_n) = P_i \wedge P_j$.

.

Chapter 6

Running a LLVM Pass

6.1 Installing Clang

First install Clang using **'sudo apt-get install'** command. Also, make sure that its version is compatible with the version of LLVM to be used.

Note - Install clang-8 (using **'sudo apt-get install clang-8'**) for working with LLVM-8.0.1, which is the current version.

6.2 Building LLVM from source

First ensure that **cmake** is installed on the system. For any further help on building LLVM from source, see the LLVM documentation page.

- First download the LLVM source code from LLVM download page or use this link to download source code for LLVM-8.0.1.

- Extract the LLVM source from the tar-package, at some preferable location. The root folder of the source will now be referred to as **LLVMsrc**.

- Create a new directory, which would be used for building the LLVM source. This directory would be referred to as **LLVMbuild**.

- Run **'cmake LLVMsrc'** from the *LLVMbuild* directory. CMake will detect the development environment, perform a series of tests, and generate the files required for building LLVM.

- Run **'cmake --build .'** from the *LLVMbuild* directory to build the source.

  Note - This step might take hours to finish. Also, building has very high memory requirements so it might also fail. In this case repeat the last step and *cmake* would detect the packages it has already built in the previous run, and start from where it was interrupted.

6.3 Getting LLVM IR using Clang

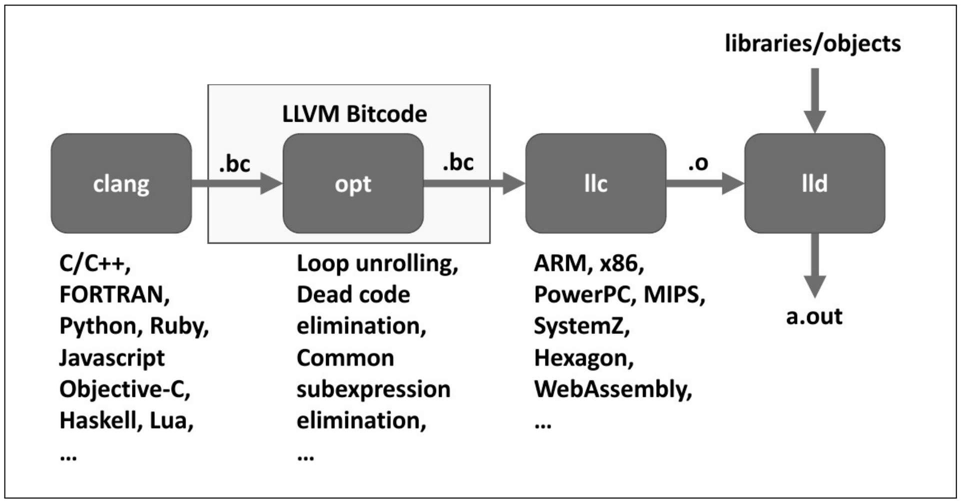
libraries/objects
LLVM Bitcode
clang
.bc
opt
.bc
llc
.o
lld
a.out
C/C++, FORTRAN, Python, Ruby, Javascript Objective-C, Haskell, Lua, ...
Loop unrolling, Dead code elimination, Common subexpression elimination, ...
ARM, x86, PowerPC, MIPS, SystemZ, Hexagon, WebAssembly, ...

Fig. 6.1 Various stages of compilation using Clang

To compile a C file named *hello.c* and get its *LLVM intermediate representation* (IR) use the following commands. We can also give the name of the output file in both of the below commands using ***'-o'*** flag.

- **clang -emit-llvm -c hello.c** - This will give LLVM bitcode in binary format file *hello.bc*.

- **clang -emit-llvm -S hello.c** - This will give LLVM code in text format file *hello.ll*.

Note - Sometimes we may also need to specify the version along with **clang** (eg. clang-8), while running commands.
We can also interconvert *.bc and *.ll file.

- **LLVMbuild/bin/llvm-as hello.ll** - To convert *hello.ll* to *hello.bc*
- **LLVMbuild/bin/llvm-dis hello.bc** - To convert *hello.bc* to *hello.ll*

We can also directly run *.bc or *.ll files as **'LLVMbuild/bin/lli hello.bc'** or **'LLVMbuild/bin/lli hello.ll'** respectively.

6.4 Writing a pass

In this section we will create a simple pass named *HelloPass*.

- Create directory *'LLVMsrc/lib/Transforms/HelloPass'*. This directory will contain the files related to our pass.
- Create a new file named *helloPass.cpp* inside *HelloPass* directory. This file will contain the code for our pass.

```
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

namespace {
  struct helloPass : public FunctionPass {
    static char ID;
    helloPass() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Function Name: ";
```

```
        errs().write_escaped(F.getName()) << '\n';
        errs() << "==============================================\n";
        for(auto bb = F.begin(); bb != F.end(); bb++){
          errs() << "\tBasicBlock Name = " << bb->getName() << "\n";
          errs() << "\tBasicBlock Size = " << bb->size() << "\n";
          for(auto i = bb->begin(); i != bb->end(); i++){
            errs() << "\t" << "Instruction: " << *i << "\n";
            errs() << "\t" << "OpCode: " << i->getOpcode() << "\n";
            errs() << "\t" << "OpCodeName: " << i->getOpcodeName() << "\n";
            errs() << "\t" << "IsBinaryOp: " << i->isBinaryOp() << "\n";
            errs() << "\t" << "IsCommutative: " << i->isCommutative() << "\n";
            errs() << "\t" << "IsAssociative: " << i->isAssociative() << "\n";
          }
          errs() << "\n\n";
        }
        return false;
      }
    };
}
char itrinstBB::ID = 0;
static RegisterPass<helloPass> X("hello",
                                 "Iterates instructions in a function");
```

The above code contains a function pass - which means the pass is run on every function defined in a file. Using iterators it traverses each basic block of the function, and for each basic block, it traverses each instruction and prints the details of the instruction - like its opcode, whether it is commutative and associative etc.

Important - Notice the first argument **hello** which is passed in the last line while registering the pass. This argument will be passed as a flag to the *HelloPass* pass when we want to run the function-pass defined inside *helloPass* structure (the template arguments in the last line).

- Create a file named *CMakeLists.txt* in the same HelloPass dirctory. This file will be used by **make** when building the pass.

```
add_llvm_library( LLVMhelloPass MODULE
   helloPass.cpp
   PLUGIN_TOOL
   opt
)
```

  LLVMhelloPass in the first line specifies the filename (with *.so extension) inside *LLVMbuild/lib/* directory, which will contain our pass when built. *helloPass.cpp* in the second line specifies the file which contains the source code for our pass.

- Add the following line to the file *'LLVMsrc/lib/Transforms/CMakeLists.txt*

```
add_subdirectory(HelloPass)
```

  This is the name of folder which contains our pass and will be used by **make** while building.

Note - More detailed tutorial on writing a pass can be found here.

6.5 Running a pass

- Firstly rebuild the LLVM source so that it includes the pass we have added. For this change current directory to *LLVMbuild* and run ***make***.

- Now run the pass on an LLVM bitcode file *helloWorld.ll* or *helloWorld.bc* as **'./bin/opt -load ./lib/LLVMhelloPass.so -hello hello.ll -o helloN.bc'**
  Notice that *LLVMhelloPass* is the name that we specified in the *CMakeLists.txt* file of our pass folder. Also, *-hello* flag is the name by which we registered our pass in the end of the file *helloPass.cpp*

Chapter 7

Implementation

This chapter presents a pseudocode of the algorithm mentioned in [6] for Herbrand Equivalence computation. The pseudocode is written taking C++ into consideration. The actual implementation done for LLVM compiler framework can be found in this github repository.

Algorithm 1 Data Structure

```
struct **IDstruct** {
    string ftype
    int parentCnt
    IDstruct *left
    IDstruct *right;

    IDstruct() {
        ftype ← ””
        parentCnt ← 0
        left ← nullptr
        right ← nullptr
    }

    IDstruct(string ftype, IDstruct *left, IDstruct *right) {
        (this→ftype) ← ftype
        (this→parentCnt) ← 0
        (this→left) ← left
        (this→right) ← right
    }
}
typedef vector<IDstruct *> Partition
map<Instruction, Partition> partitions
```

The objects of structure *IDstruct* would be created dynamically using *new* operator. The convention taken is that any variable which is assigned a value of type *IDstruct*, would actually be assigned a pointer to a dynamically created object of *IDstruct*. Also, if any variable points to an object of *IDstruct*, then its *parentCnt* is incremented by 1. Also, when a variable which was earlier pointing to an object of *IDstruct* stops pointing to it, the *parentCnt* is decremented by 1. Finally, the dynamically allocated memory is freed whenever the *parentCnt* of an object becomes 0. These things aren't explicitly mentioned in the pseudocode.

Partition type is a vector of pointers to object of type IDstruct. Each index of such vector corresponds to a program expression of length atmost 2, which is fixed arbitrarily in the beginning. Expressions pointing to same IDstruct object are equivalent (belong to same partition) at that program point, which can be determined by checking for pointer equivalence. *partitions* is a map from the instructions in the program to a *Partition*.

Algorithm 2 Main Procedure

```
procedure Main
    partitions[$I_0$] = findInitialPartition()

    for I $\in$ (Instructions $\setminus\{I_0\}$) do
        partitions[I] = $\top$

    Bool converged = false
    while not converged do
        converged $\leftarrow$ true

        for I $\in$ Instructions do
            oldPartition $\leftarrow$ partitions[I]

            if I is function point with Predecessors(I) = {J} then
                AssignStatement(partitions[I], partitions[J], I)
            else
                Confluence(partitions[I], I)

            if oldPartition $\neq$ partitions[I] then
                converged $\leftarrow$ false
```

Here, I_0 is an imaginary instruction such that the predecessor of the first instruction of the program is I_0.

Algorithm 3 Transfer Function

```
procedure AssignStatement(Partition &curPart, Partition &prevPart, Instruction I)
    curPart = prevPart

    if I is (z := x) then
        curPart[z] ← curPart[x]
    else if I is (z := x op y) then
        ptr ← exists({op, curPart[x], curPart[y]})
        if ptr == nullptr then
            ptr ← IDstruct(op, curPart[x], curPart[y])
        curPart[z] ← ptr

    x ← LValue(I)
    for op ∈ Operators do
        for y ∈ (Constants ∪ Variables) do
            ptr ← exists({op, curPart[x], curPart[y]})
            if ptr == nullptr then
                ptr ← IDstruct(op, curPart[x], curPart[y])
            curPart[x op y] ← ptr

            ptr ← exists(op, curPart[y], curPart[x])
            if ptr == nullptr then
                ptr ← IDstruct(op, curPart[y], curPart[x])
            curPart[y op x] ← ptr
```

Instructions, *Constants*, *Variables* and *Operators* represent the set of instructions, constants, variables and operators occuring in the program respectively. *Terms* represents all expressions of length exactly 2. *Predecessor* function gives the list of predecessors of a instruction in the program control flow graph. Finally, *exists* is a function that accepts a tuple with fields $\{string, IDstruct*, IDstruct*\}$ and tells whether an IDstruct object exists with *ftype* as the first element of the tuple and *left, right* as second and third element of the tuple respectively. If so, it returns a pointer to the object otherwise returns nullptr.

Algorithm 4 Confluence Function

```
procedure Confluence(Partition &curPart, Instruction I)
    Partition tempPartition
    Bool accessFlag

    for x $\in$ (Constants $\cup$ Variables) do accessFlag[x] $\leftarrow$ false

    for x $\in$ (Constants $\cup$ Variables) do
        if not accessFlag[x] then
            accessFlag[x] $\leftarrow$ true
            if $\forall$ I' $\in$ Predecessors(I), partition[I'][x] are same then
                tempPartition[x] $\leftarrow$ partition[I'][x]
            else
                ptr $\leftarrow$ IDstruct()
                tempSet $\leftarrow$ intersection(getClass(x, partition[I']), $\forall$ I' $\in$ Predecessors(I))
                for y $\in$ (tempSet $\cap$ (Constants $\cup$ Variables)) do
                    accessFlag[y] = true
                    tempPartition[y] = ptr
    for (x op y) $\in$ Terms do
        ptr $\leftarrow$ exists({op, curPart[x], curPart[y]})
        if ptr == nullptr then
            ptr $\leftarrow$ IDstruct(op, curPart[x], curPart[y])
        tempPartition[x op y] $\leftarrow$ ptr

    AssignStatement(curPart, tempPartition, I)
```

Algorithm 5 Finding Initial Partition

```
procedure initialPartition
    Partition partition
    for x $\in$ (Constants $\cup$ Variables) do
        partition[x] $\leftarrow$ IDstruct()
    for (x op y) $\in$ Terms do
        partition[x op y] $\leftarrow$ IDstruct(op, partition[x], partition[y])
    return partition
```

Algorithm 6 Finding equivalence class of an expression

```
procedure getClass(Term/Variable/Constant z, Partition curPart)
    set equivalenceClass
    for x $\in$ (Constants $\cup$ Variables $\cup$ Terms) do
        if curPart[z] == curPart[x] then
            equivalenceClass.insert(x)
    return equivalenceClass
```

Chapter 8

Conclusion and Future Work

Reading the concerned paper gave theoretical insights for the relevant problem. After having the initial working implementation of the algorithm, the next task is to optimize it for better performance and then use it for optimzation of actual programs. Once this is done we can proceed to benchmark it against standards. Finally, a proof that the algorithm indeed is able to find all the Herbrand equivalent expressions restricted to program constants, variables and operators and thus adhers with the theory, has to be proposed.

References

[1] G. C. Necula, "Translation validation for an optimizing compiler," *In Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pp. 83–94, 2000.

[2] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," *In B. Steffen, Editor, Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference*, vol. LNCS 1384, pp. 151–166, 1998.

[3] O. Ruthing, J. Knoop, and B. Steffen, "Detecting equalities of variables: Combining efficiency with precision," *In 6th International Symposium on Static Analysis*, pp. 232–246, 1999.

[4] S. Gulwani and G. C. Necula, "A polynomial time algorithm for global value numbering," *In Science of Computer Programming 64*, pp. 97–114, January 2007.

[5] S. Nabeezath and V. Paleri, "A polynomial time algorithm for global value numbering," *CoRR*, pp. 1–11, 6th April, 2018.

[6] J. Babu, K. M. Krishnan, and V. Paleri, "A fix point characterization of herbrand equivalence of expressions in data flow frameworks," *In 18th Indian Conference on Logic and its Applications*, 5th March, 2019.

[7] G. A. Kildall, “A unified approach to global program optimization,” *In 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 194–206, October, 1973.

[8] B. Alpern, M. N. Wegman, and F. K. Zadeck, “Detecting equality of variables in programs,” *In 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 1–11, 1998.

[9] O. Ruthing, J. Knoop, and B. Steffen, “Detecting equalities of variables: Combining efficiency with precision,” *In Static Analysis Symposium*, vol. 1694, pp. 232–247, 1999.