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

CERTIFICATE

*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

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 **Herbrand Equivalence** (see section 1.1). 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 (see 1.2example below).

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.

1.2 A Simple Example

Figure 1.2 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 $+$.

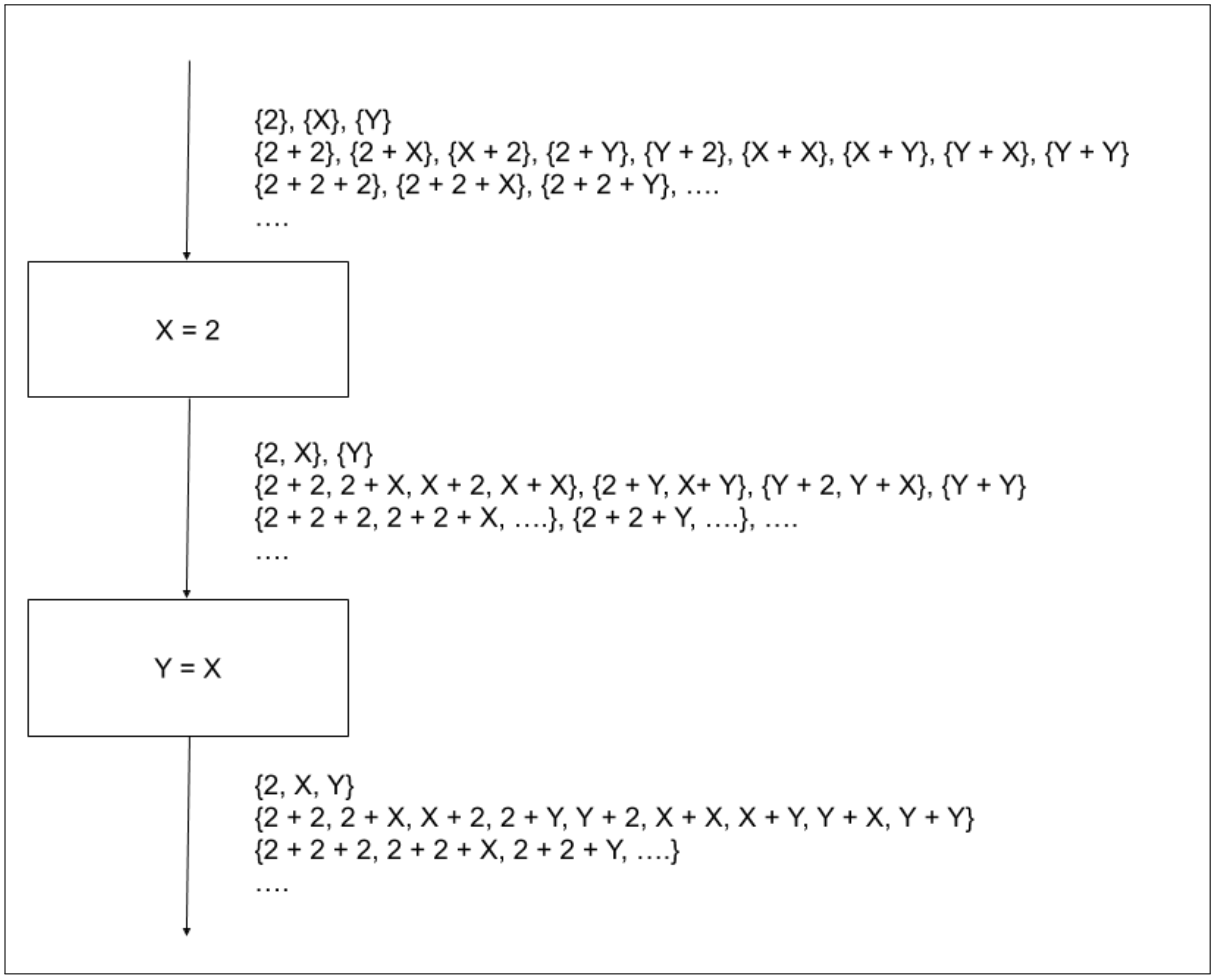

Fig. 1.1 Example of Herbrand Equivalence

- After assignment $X = 2$, any occurrence of X in an expression can be replaced with 2. So, now all expressions with 2 in place of X and vice versa are equivalent - that means $2 + 2$, $2 + X$, $X + 2$, $X + X$ are all equivalent - this still is just syntactic information because X and 2 are equivalent. However, if 4 was also in the universe of expressions, $2 + 2$ and 4 are not equivalent as this is semantic information of $+$.

- After assignment $Y = X$, any occurrence of Y in an expression can be replaced with X. Because X and 2 are already equivalent, it means now 2, 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.

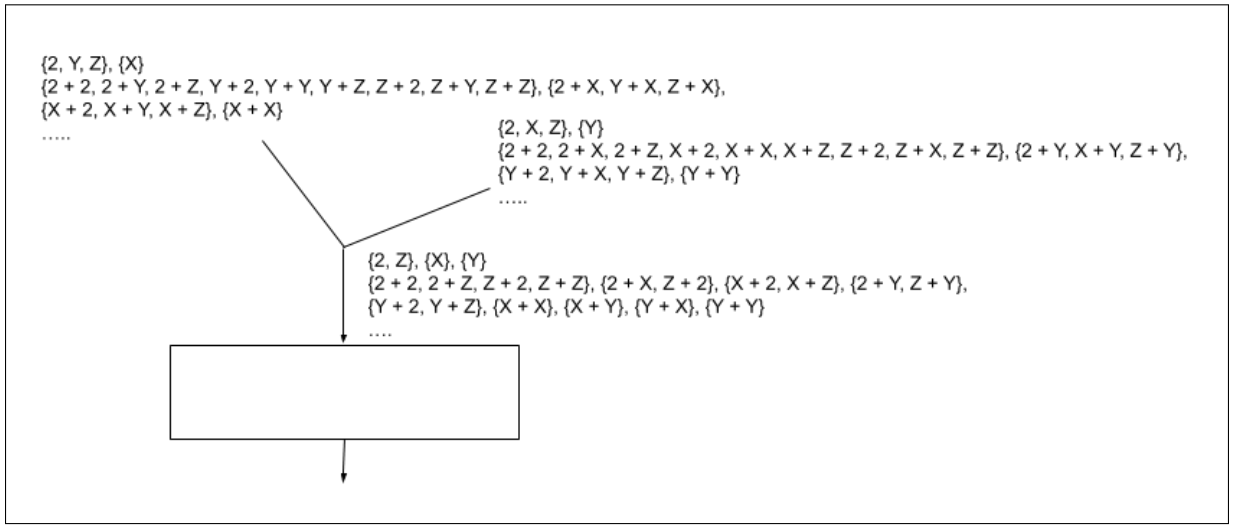

Fig. 1.2 Example of Herbrand Equivalence analysis at a confluence point

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 which are interconvertible by replacement of any of these three, with any other.
- The case with the right branch is similar, except X is equivalent to 2 and Z instead of Y.
- 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.

1.3 Goal of the Project

[4] gives an algorithm for Herbrand Equivalence analysis restricted to program expressions. The basic goal of this project is to refine this general algorithm and then implement it for Clang-LLVM compiler; then extend this algorithm to use the analysis information for performing actual program optimizations. Finally, a proof of correctness of the algorithm has to be presented which would also be based on the work in [4].

1.4 Organization of the Report

Chapter 2 gives a brief overview of previous works related to the Herbrand Equivalence; then summary of the papers [5, 6, 4] are specifically presented. Chapter 6 provides a tutorial on writing an LLVM optimzation pass. Chapter 7 gives pseudocode for Herbrand Equivalence analysis of a program; algorithms in chapter 8 extends these to use the analysis information for performing actual program optimizations.

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 [5] 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 operators occur 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 [6] 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 [4] 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 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, they 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 can be represented as

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

where, c and x are constants and variables occurring in the program respectively. Any expression of length greater than two (in terms of number of operands) can be converted into two length expression by introduction of extra variables.

The data structure used is called **Strong Equivalence DAG (SED)**. Each node of SED 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'.

For detailed implementation of **Join** and **Assignment** functions, as well as a correctness proof of the algorithm, see [5].

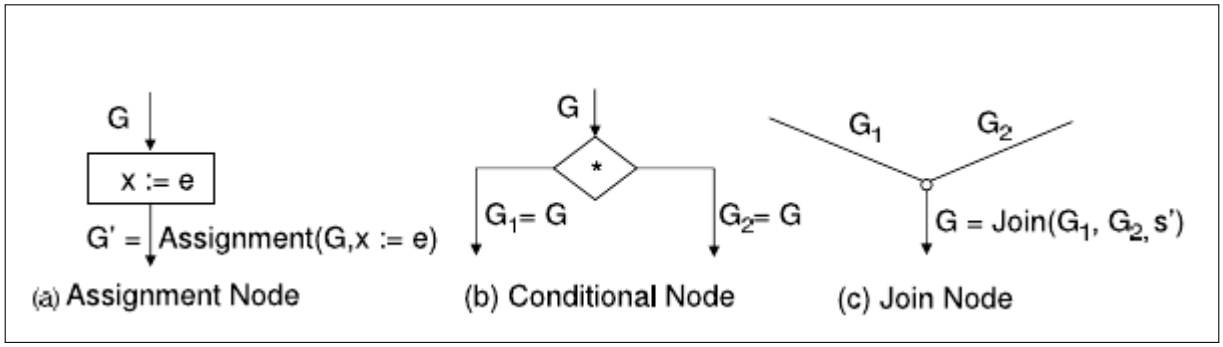

Fig. 3.1 Computation of SED for flowgraph nodes of a program

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. 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 [6].

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 [4] 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 [4].

5.1 Program Expressions

Let $\mathcal{C}$ and $\mathcal{X}$ 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 $c \in \mathcal{C}$ and $x \in \mathcal{X}$.

5.2 Congruence Relation

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

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

Let $\mathcal{G}(\mathcal{T})$ be the set of all congruences over $\mathcal{T}$. We define an order, $\mathcal{P}_1 \preceq \mathcal{P}_2$ for $\mathcal{P}_1, \mathcal{P}_2 \in \mathcal{G}(\mathcal{T})$, if $\forall \mathcal{A}_1 \in \mathcal{P}_1, \ \exists \mathcal{A}_2 \in \mathcal{P}_2$ such that $\mathcal{A}_1 \subseteq \mathcal{A}_2$.
We define binary **confluence operation** on $\mathcal{G}(\mathcal{T})$ as

$$\mathcal{P}_1 \wedge \mathcal{P}_2 \ = \ \{\mathcal{A}_i \cap \mathcal{B}_j \mid \mathcal{A}_i \in \mathcal{P}_1 \text{ and } \mathcal{B}_j \in \mathcal{P}_2\}$$

Now, we extend $\mathcal{G}(\mathcal{T})$ to $\overline{\mathcal{G}(\mathcal{T})}$ by introducing abstract congruence $\top$ satisfying $\mathcal{P} \wedge \top = \top, \forall \mathcal{P} \in \overline{\mathcal{G}(\mathcal{T})}$. Also, we denote the congruence in which every element is in a separate class as $\bot$.
With these definitions, $(\overline{\mathcal{G}(\mathcal{T})}, \preceq, \bot, \top)$ forms a complete lattice, with $\wedge$ as its **meet operator**.

5.3 Transfer function

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

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

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

5.4 Non deterministic assignment

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

- $t \cong_{\mathcal{P}} t'$
- $\forall \beta \in (\mathcal{T} \setminus \mathcal{T}(y)),\ t[y \leftarrow \beta] \cong_{\mathcal{P}} t'[y \leftarrow \beta]$

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

5.5 Dataflow analysis Framework

A dataflow framework over $\mathcal{T}$ is $\mathcal{D} = (G, \mathcal{F})$ where $G(V, E)$ is the control flow graph associated with the program and $\mathcal{F} = \{h_k : k \in V$ is a **transfer program point**$\}$ is **a collection of transfer functions**.

5.6 Herbrand Congruence Function

The Herbrand Congruence function $\mathcal{H}_{\mathcal{D}} : V(G) \rightarrow \overline{\mathcal{G}(\mathcal{T})}$ gives the Herbrand Congruence associated with each program point and is defined to be **the maximum fix point** of the **continuous composite transfer function** $f_{\mathcal{D}} : \overline{\mathcal{G}(\mathcal{T})}^n \rightarrow \overline{\mathcal{G}(\mathcal{T})}^n$, where $\overline{\mathcal{G}(\mathcal{T})}^n$ is the product lattice, $f_{\mathcal{D}}$ is a function satisfying $\pi_k \circ f_{\mathcal{D}} = f_k$. Here π_k is the projection map and $f_k : \overline{\mathcal{G}(\mathcal{T})}^n \rightarrow \overline{\mathcal{G}(\mathcal{T})}$ 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 $\pi_{i,j} : \overline{\mathcal{G}(\mathcal{T})}^n \rightarrow \overline{\mathcal{G}(\mathcal{T})}$ is given by $\pi_{i,j}(P_1, \ \ldots, \ P_n) = P_i \wedge P_j$.

.

Chapter 6

Writing 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

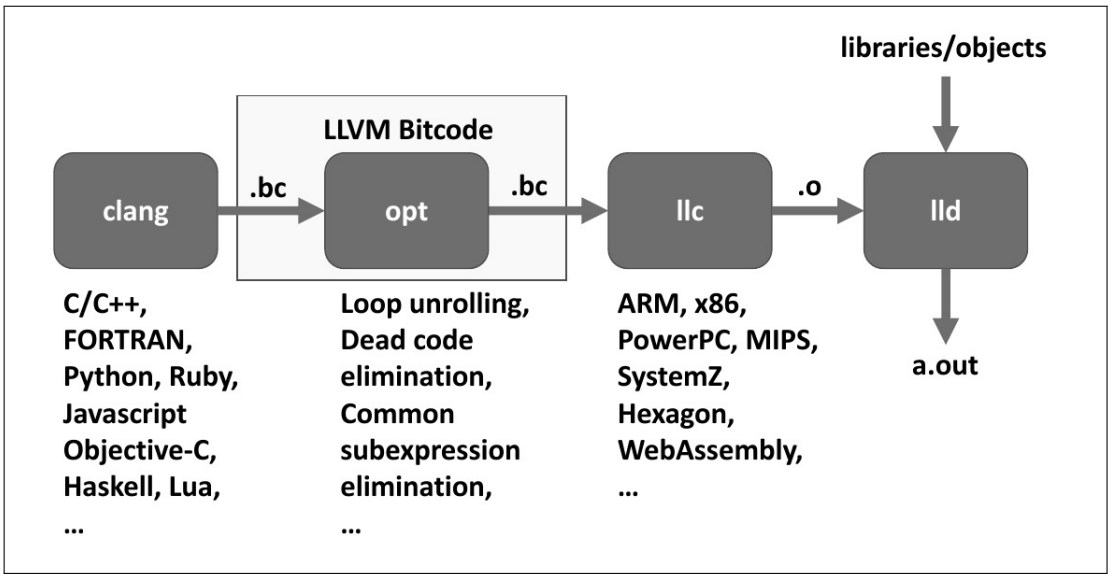

Fig. 6.1 Various stages of compilation using Clang

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 which is given below.

```
1  #include "llvm/ADT/Statistic.h"
2  #include "llvm/IR/Function.h"
3  #include "llvm/Pass.h"
4  #include "llvm/Support/raw_ostream.h"
5  using namespace llvm;
6
7  namespace {
8    struct helloPass : public FunctionPass {
9      static char ID;
10     helloPass() : FunctionPass(ID) {}
11
```

```
12      bool runOnFunction(Function &F) override {
13        errs() << "Function Name: ";
14        errs().write_escaped(F.getName()) << '\n';
15        errs() << "================================================\n";
16        for(auto bb = F.begin(); bb != F.end(); bb++){
17          errs() << "\tBasicBlock Name = " << bb->getName() << "\n";
18          errs() << "\tBasicBlock Size = " << bb->size() << "\n";
19          for(auto i = bb->begin(); i != bb->end(); i++){
20            errs() << "\t" << "Instruction: " << *i << "\n";
21            errs() << "\t" << "OpCode: " << i->getOpcode() << "\n";
22            errs() << "\t" << "OpCodeName: " << i->getOpcodeName() << "\n";
23            errs() << "\t" << "IsBinaryOp: " << i->isBinaryOp() << "\n";
24            errs() << "\t" << "IsCommutative: " << i->isCommutative() << "\n";
25            errs() << "\t" << "IsAssociative: " << i->isAssociative() << "\n";
26          }
27          errs() << "\n\n";
28        }
29        return false;
30      }
31    };
32  }
33  char itrinstBB::ID = 0;
34  static RegisterPass<helloPass> X("hello",
35                                   "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.

```
1 add_llvm_library( LLVMhelloPass MODULE
2   helloPass.cpp
3   PLUGIN_TOOL
4   opt
5 )
```

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*

```
1 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 *source.ll* or *source.bc* as
  './bin/opt -load ./lib/LLVMhelloPass.so -hello source.bc -o sourceN.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

Algorithm for Herbrand Equivalence

This chapter presents a pseudocode of the algorithm mentioned in [4] for Herbrand Equivalence analysis. The pseudocode is written taking C++ into consideration. The actual implementation done for LLVM compiler framework can be found in this github repository.

Objects of structure **IDstruct**, will be used to hold information about the equivalence of program expressions. *Partition* is vector type, whose each index corresponds to either a constant, a variable or length two expression (from now on expression would mean only these three). It is assumed that *Partition* can be indexed directly with expressions, instead of indexing with integers. This can be easily taken care of in an actual program by fixing indexes for the expressions at the beginning, for the whole duration of the program. There will be a *Partition* for each program point (*partitions* map) and at a given program point, expressions which holds pointer to same $IDstruct$ object would be Herbrand equivalent at that program point. So, *Partition* actually represents a partition which justifies its name.

The first three fields of $IDstruct$ object are relavant only for two length expressions. For expression $(x\ op\ y)$ at instruction I, these fields would be op, $partitions[I][x]$, $partitions[I][y]$ respectively. For a constant or a variable these would be set as $''$, $nullptr$, $nullptr$ respectively. The field $isConst$ is true only if the $IDstruct$ object represents a constant expression

Algorithm 1 Data Structure

```
1: struct IDstruct {
2:     char opSymbol
3:     IDstruct* leftID
4:     IDstruct* rightID
5:     bool isConst
6:     int constVal
7:
8:     IDstruct() {                          ▷ To create IDstruct object for a variable
9:         this.opSymbol ←″
10:        this.leftID ← nullptr
11:        this.rightID ← nullptr
12:        this.isConst ← false
13:        this.constVal ← 0
14:    }
15:
16:    IDstruct(int constVal) {              ▷ To create IDstruct object for a constant
17:        this.opSymbol ←″
18:        this.leftID ← nullptr
19:        this.rightID ← nullptr
20:        this.isConst ← true
21:        this.constVal ← constVal
22:    }
23:
24:    IDstruct(string op, IDstruct* leftID, IDstruct* rightID) {
25:                                  ▷ To create IDstruct object for a length two expression
26:        this.opSymbol ← op
27:        this.leftID ← leftID
28:        this.rightID ← rightID
29:
30:        if (leftID.isConst && rightID.isConst) {
31:            this.isConst ← true
32:            this.constVal ← Evaluate(op, leftID.constVal, rightID.constVal)
33:        } else {
34:            this.isConst ← false
35:            this.constVal ← 0
36:        }
37:    }
38: }
39:
40: typedef vector<IDstruct *> Partition
41: map<Instruction, Partition> partitions
```

and in that case $constVal$ will hold the corresponding value of the expression. These two fields are not required for Herbrand Equivalence analysis, but would be useful later during optimzations (specifically for constant propagation and constant folding).

Instructions, **Constants**, **Variables** and **Operators** represent the set of instructions, constants, variables and operators occuring in the program respectively. I_0 is an imaginary instruction which is predecessor of the first instruction. **Expressions** is the set of relavant program expressions (of length atmost two).

Also note the following functions whose details have not been provided.

Evaluate($op,\ x,\ y$) :- Returns result of ($x\ op\ y$).

Instersection :- Simple set intersection.

FindIDstruct($op,\ leftID,\ rightID$) :- If there exists an $IDstruct$ object whose first three fields are op, $leftID$, $rightID$ respectively then it returns pointer to it, otherwise calls its third constructor to first create a new object and then returns a pointer to it.

Algorithm 2 Herbrand Equivalence Analysis

```
1:  procedure HerbrandAnalysis
2:      partitions[I_0] ← findInitialPartition()
3:
4:      for I ∈ Instructions do
5:          partitions[I] ← ⊤
6:
7:      Bool converged ← false
8:      while not converged do
9:          converged ← true
10:
11:         for I ∈ Instructions do
12:             Partition oldPartition ← partitions[I]
13:
14:             if I is function point with Predecessors(I) = {J} then
15:                 TransferFunction(partitions[I], partitions[J], I)
16:             else
17:                 Partition confPartition ← ConfluenceFunction(I)
18:                 TransferFunction(paritions[I], confPartition, I)
19:
20:             if not SamePartition(oldPartition, partitions[I]) then
21:                 converged ← false
```

Algorithm 3 Finding Initial Partition

```
1:  procedure FindInitialPartition
2:      Partition partition
3:
4:      for x ∈ Constants do
5:          partition[x] ← IDstruct(x)
6:      for x ∈ Variables do
7:          partition[x] ← IDstruct()
8:      for (x op y) ∈ Expressions do
9:          partition[x op y] ← FindIDstruct(op, partition[x], partition[y])
10:
11:     return partition
```

Algorithm 4 Transfer Function

```
1:  procedure TransferFunction(Partition& curPart, Partition& prevPart, Instruction I)
2:
3:      curPart ← prevPart
4:
5:      if I is z := x then
6:          curPart[z] ← curPart[x]
7:      else if I is z := x op y then
8:          curPart[z] ← FindIDstruct(op, curPart[x], curPart[y])
9:
10:     x ← LValue(I)
11:     for op ∈ Operators do
12:         for y ∈ (Constants ∪ Variables) do
13:             curPart[x op y] = FindIDstruct(op, curPart[x], curPart[y])
14:             curPart[y op x] = FindIDstruct(op, curPart[y], curPart[x])
```

Algorithm 5 Confluence Function

```
1: procedure ConfluenceFunction(Instruction I)
2:     Partition confPartition
3:     vector < bool > accessFlag
4:
5:     for x ∈ (Constants ∪ Variables) do accessFlag[x] ← false
6:
7:     for x ∈ (Constants ∪ Variables) do
8:         if not accessFlag[x] then
9:             accessFlag[x] ← true
10:            if ∀I' ∈ Predecessors(I), partitions[I'][x] is same then
11:                confPartition[x] ← partitions[I'][x]
12:            else
13:                ptr ← IDstruct()
14:                tempSet ← Intersection(GetClass(x, partition[I']), ∀I' ∈Predecessors(I))
15:                for y ∈ (tempSet ∩ (Constants ∪ Variables)) do
16:                    accessFlag[y] ← true
17:                    confPartition[y] ← ptr
18:
19:    for (x op y) ∈ Expressions do
20:        confPartition[x op y] ← FindIDstruct(op, confPartition[x], confPartition[y])
21:
22:    return confPartition
```

Algorithm 6 To check if two partitions are same

```
1: procedure SamePartition(Partition first, Partition second)
2:     for x ∈ Expressions do
3:         if GetClass(first, x) ≠ GetClass(second, x) then
4:             return false
5:     return true
```

Algorithm 7 Finding equivalence class of an expression in a partition

```
1: procedure GetClass(Partition partition, Expression exp)
2:                 ▷ Expression represents constants, variables and length two epxressions
3:     set<Expression> equivalenceClass
4:     for x ∈ Expressions do
5:         if partition[exp] == partition[x] then
6:             equivalenceClass.insert(x)
7:     return equivalenceClass
```

Chapter 8

Performing Optimizations

This chapter explains how to use the Herbrand Equivalence analysis information to perform actual program optimizations.

8.1 Available Variables

A variable is said to be **available** at a basic block if it is defined at some point along all the execution paths from the start of the program that reaches the beginnning of that basic block. It is forward data analysis problem and can be determined via fixed point computation.

The universe $\mathcal{U}$ would be the set of all program variables. Also, let $\mathcal{B}$ be the set of all basic blocks in the program. Denote by $DEF[B]$ the set of variables defined in the basic block $B \in \mathcal{B}$, and $IN[B]$ be corresponding set of available variables. Initialise $OUT[B] = \mathcal{U}, \forall B \in \mathcal{B}$ and perform updates as follows till a fixed point is reached.

$$IN[B] = \cap_{B' \in Predecessors(B)} OUT[B']$$

$$OUT[B] = IN[B] \cup DEF[B]$$

Note that the definition of **available variables** is different from that of **reaching**

definitions and **available expressions**. If a variable is available at the beginning of a given basic block, then that variable can be used in the basic block without defining it because by *definition of avilable variables* it would already be defined along every path from the start of the program to that basic block.

The definition of available variables can be extended to that of any instruction. Variables available at an instruction I is the union of variables available at the beginning of its basic block and any variables defined by the instructions preceding it in that block.

8.2 Performing Optimizations

The results of Herbrand Equivalence and available variables analysis can be used to perform optimizations involving redundant expression elimination and some dead code elimination.

Suppose, there is an instruction $I : z \leftarrow e$, where e is an expression (as already mentioned expression means either a constant, or a variable or a length two expression). First find the Herbrand equivalence class of z at that program point. If the class represents a constant valued expression ($partitions[I][z].isConst == true$), then this instruction can be deleted and all uses of z replaced by $partitions[I][z].constVal$, till z is redefined. Else if it is not a constant valued expression, $\mathcal{S} = (getClass(partitions[I], z) \cap IN[I])$ will be the set of variables, such that z can be replaced by $s \in \mathcal{S}$ (before z and s are redefined), without changing the meaning of the program.

Pseudocode 8 is the precise formulation of the mentioned facts.

Algorithm 8 Performing Optimizations

```
1: procedure OPTIMIZE
2:     for B ∈ ℬ do
3:         Insts ← Instructions(B)
4:         AvailVars ← IN[B]
5:
6:         for I ∈ Insts do
7:             Insts.remove(I)
8:             z ← LValue(I)
```

```
9:          if Defined(z) then
10:             IDstruct* ptr ← partitions[I][z]
11:
12:             if ptr.isConst then
13:                 Bool reachedEnd ← true
14:                 for I' ∈ Insts do
15:                     if LValue(I') == z then
16:                         reachedEnd ← false
17:                         break
18:                     Replace(I', z, ptr.constVal)
19:                 if reachedEnd then
20:                     B.insert(z ← ptr.constVal)
21:
22:                 DeleteInstruction(I)
23:             else
24:                 set<Expressions> curPart ← GetClass(partitions[I], z)
25:                 curPart ← (curPart ∩ AvailVars)
26:
27:                 if not curPart.empty() then
28:                     Bool reachedEnd ← true
29:                     replacement ← curPart.first()
30:
31:                     for I' ∈ Insts do
32:                         if LValue(I') == z then
33:                             reachedEnd ← false
34:                             break
35:                         curPart = curPart \ {LValue(I')}
36:                         if not curPart.empty() then
37:                             replacement ← curPart.first()
38:                             Replace(I', z, replacement)
39:                         else
40:                             B.insert(I', z ← replacement)
41:                             AvailVars ← AvailVars ∪ {z}
42:                             reachedEnd ← false
43:                             break
44:
45:                     if reachedEnd then
46:                         B.insert(z ← replacement)
47:                     DeleteInstruction(I)
48:
49:                 else
50:                     AvailVars ← AvarilVars ∪ {z}
```

Also note the following functions whose details has not been provided.

Instructions(B) :- Returns a list of instructions in the basic block B.

Replace$(I', z, replacement)$:- Replaces variable z in the instruction I' with *replacement* which can be a constant or another variable.

B.insert(I) :- Inserts instruction I, at the end of basic block B.

B.insert(I, I') :- Insert instruction I' before instruction I in basic block B.

DeleteInstruction(I) :- Deletes instruction I from the program.

set.empty() :- Returns $true$ if *set* is empty, else $false$.

set.first() :- Returns first element from *set*.

8.3 LLVM Implementation

The LLVM specific implementation of the pseudocodes mentioned can be found in this github repository.

Chapter 9

Conclusion and Future Work

Reading the concerned paper gave theoretical insights for the relevant problem. Currently, implementation of the algorithms for Herbrand Equivalence analysis and using it for performing optimizations, for Clang/LLVM compiler framework has been finished. The next task is to prove the correctness of the algorithm. For this, we are trying to modify the algorithm a bit for convinience of proof. So, we are concurrently working on the proof and implementation of this new algorithm.

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] 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.

[5] 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.

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

[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.