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 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 perform multiple optimizations 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.

A typical modern day compiler 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 semantic 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

earlier 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 given 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 example in Section 1.2).

For **Herbrand equivalence analysis**, the universe is 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 it such that two expressions are Herbrand equivalent at that point if and only if they belong to the same partition class of that point.

1.2 A Simple Example

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 $+$.

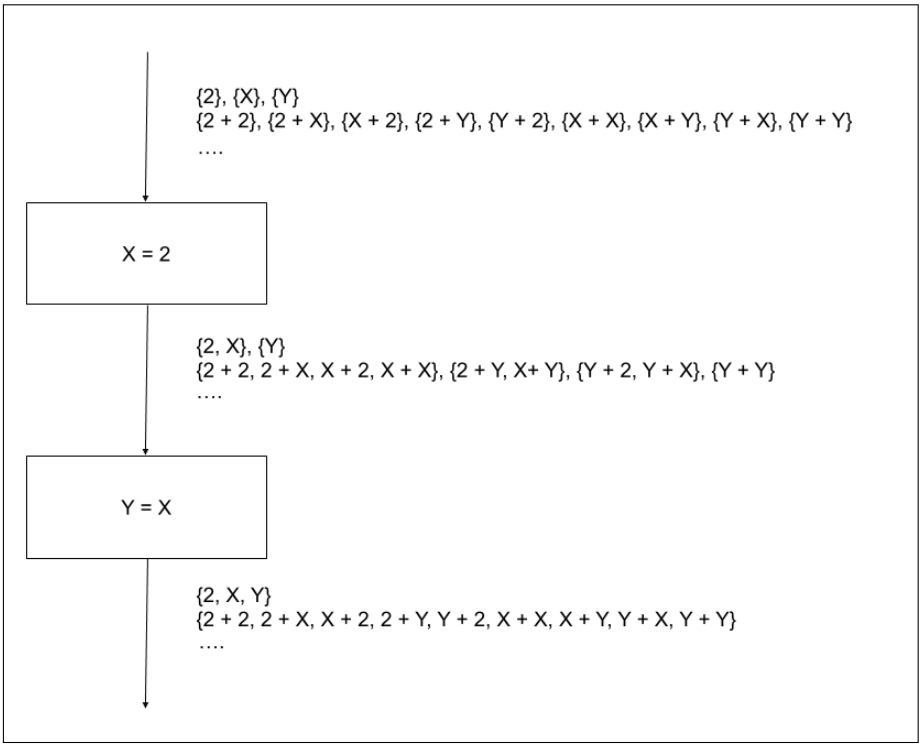

Figure 1.1 Example of Herbrand Equivalence

- After assignment $X = 2$, any occurrence of X in an expression can equally 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 is also in the universe of expressions, $2 + 2$ and 4 are not equivalent as this is semantic information about operator $+$.

- 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 three with any of the other two. For this example, it means that any two expressions of the same length are equivalent.

Figure 1.2 shows what happens at a **confluence point** - a point where multiple execution paths meet. Two expressions are Herbrand equivalent at the confluence point only

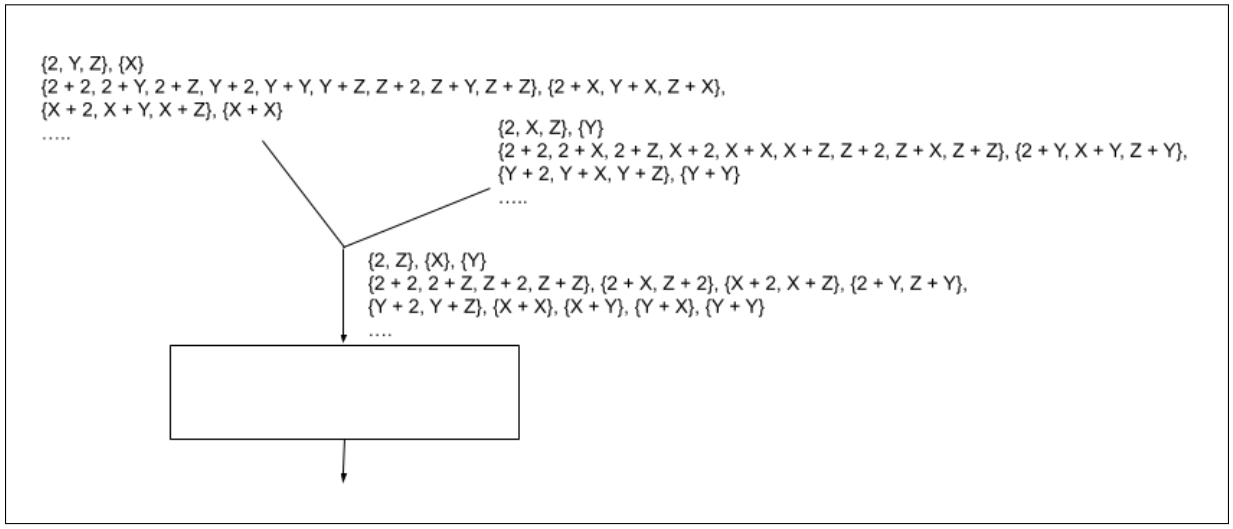

Figure 1.2 Example of Herbrand Equivalence analysis at a confluence point

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 the other two.
- 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 are equivalent at both the predecessors points. X is equivalent to 2 and Z at the right predecessor but not the left one and Y is 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

Babu, Krishnan and Paleri [4] has given 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. The implementation is also

to be benchmarked using SPEC CPU benchmarks. Finally, a proof of correctness of the algorithm based on the theoretical work of Babu et al. [4] has to be presented.

1.4 Organization of the Report

Chapter 2 gives an overview of the previous works related to Herbrand equivalence; then summary of works of Babu et al. [4] is specifically presented in Chapter 3 as it is closely related to the project. The updated algorithm for Herbrand equivalence analysis is given in Chapter 4. Chapter 5 presents some important test cases on which the Herbrand analysis algorithm can be checked for correctness. Chapter 6 discusses about Clang/LLVM compiler and also provides a tutorial for writing LLVM optimzation passes. Chapter 7 and Chapter 8 explains the implementation of the algorithm done for Clang/LLVM compiler and a toy language respectively. Finally, Chapter 9 provides the details of the attempt made for proving the correctness of the algorithm. Chapter 10 concludes the report.

Chapter 2

Review of Prior Work

Existing algorithms for Herbrand equivalence analysis are either exponential or imprecise. The precise algorithms are based on an early algorithm by Kildall [5], which discovers equivalences by performing an abstract interpretation over the lattice of Herbrand equivalences. Kildall's algorithm 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) [6] is efficient but 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) [7] 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.

2.1 Algorithm by Gulwani and Necula

Gulwani and Necula [8] 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. 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.

They showed that Herbrand equivalences among program sub expressions can always be represented using linear sized value graph. So contrasting to Kildall's algorithm, which finds **all the Herbrand equivalent classes** corresponding to constants, variables and operators occurring in the program, their 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.

They also 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.

2.1.1 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 and F stands for an operator. 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 2.1. s' in the argument of `Join` is a positive integer, and it returns equivalences between expressions of size atmost s'.

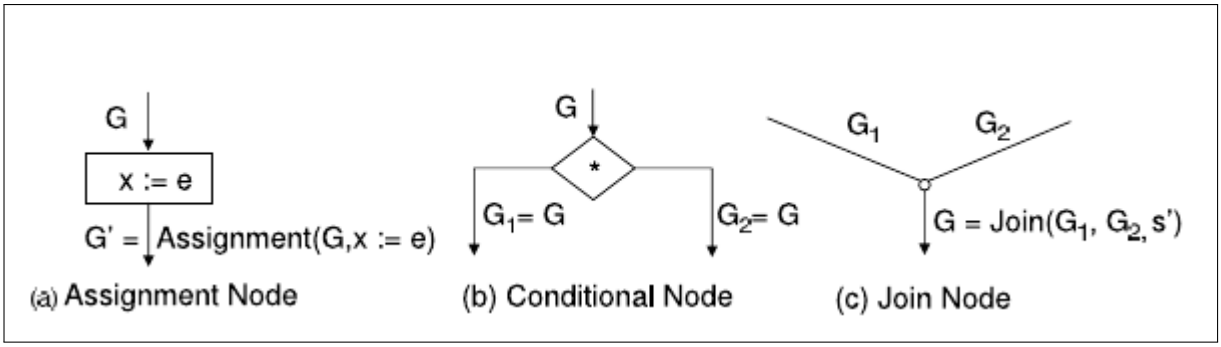

Figure 2.1 Computation of SED for flowgraph nodes of a program

For detailed implementation of `Join` and `Assignment` functions, as well as a correctness proof of the algorithm, see [8].

2.1.2 Complexity of the Algorithm

The complexity of the algorithm is $\mathcal{O}(k^3 * N * j)$, 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.

2.2 Algorithm by Saleena and Paleri

Saleena and Paleri [9] 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 one talks about equivalences at a particular program point but GVN is 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.

2.2.1 Notations

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

- $x \ ::= \ e$
- $e \ ::= \ x \ | \ c \ | \ x_1 \ op \ x_2$

The flow graph also has two additional empty `ENTRY` and `EXIT` nodes. For a node n, $\texttt{IN}_n$ and $\texttt{OUT}_n$ 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 will be considered as its first element. For a node n, $\texttt{EIN}_n$ and $\texttt{EOUT}_n$ denotes the expression pools at input and output program points of the node.

2.2.2 Value Expression

The **value expression** corresponding to an expression is obtained by replacing the 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. So 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.

2.2.3 Overview of the 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 $\mathsf{EOUT}_{\mathsf{ENTRY}} = \phi$, 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 algorithm refer to [9].

2.3 Algorithm by Babu, Krishnan and Paleri

One problem is that most of these alogrithms are 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 and

showed that this definition is equivalent to the classical meet over all path characterization over the set of all possible expressions. They also proposed an algorithm which is able to detect all the equivalences as by that of Saleena and Paleri. This is discussed in detail in Chapter 3.

2.4 Conclusion

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

Formulation by Babu, Krishnan and Paleri

One of the problems with other former approaches to Herbrand equivalence is that most of the alogrithms are 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 form 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].

3.1 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 $x \in \mathcal{X}$.

3.2 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 $\mathcal{T}$. An ordering is defined over $\mathcal{G}(\mathcal{T})$ as $\mathcal{P}_1 \preceq \mathcal{P}_2$ 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 $\mathcal{A}_1 \subseteq \mathcal{A}_2$.
Also binary **confluence operation ($\wedge$)** is defined on $\mathcal{G}(\mathcal{T})$ as

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

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

3.3 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 $\mathcal{A}_i \in \mathcal{P}$
- $f_{y:=\beta}(\mathcal{P}) = \{\mathcal{B}_i \mid \mathcal{B}_i \neq \phi\}$

This definition is extended 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**.

3.4 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 this transfer function is extended 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**.

3.5 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 $\mathcal{F}$ is a collection of transfer function associated with program points.

3.6 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 $\texttt{Pred}(k) = \{j\}$, 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 $\texttt{Pred}(k) = \{i, j\}$, 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 $\pi_{i,j}(\mathcal{P}_1, \ \ldots, \ \mathcal{P}_n) = \mathcal{P}_i \wedge \mathcal{P}_j$.

Chapter 4

Algorithm for Herbrand Analysis

This chapter presents an algorithm for Herbrand analysis. The pseudocode mentioned here is an updated version of the algorithm mentioned in [4] for Herbrand equivalence analysis. The corresponding implementation done for LLVM compiler framework and a toy language can be found on GitHub. Also, see Chapter 7 and Chapter 8 for more details.

One important distinction must be clear between the general Herbrand analysis problem and the one that the algorithm in this chapter addresses. The universe for the Herbrand equivalence problem is the set of all expressions that can be formed using constants, variables and operators used in the program. But as already mentioned, the algorithm here is concerned with a restricted universe - set of expressions of length atmost two formed using constants, variables and operators in the program.

4.1 Notation

Let $\mathcal{C}$ and $\mathcal{X}$ be the set of constants and variables used in the program; $\mathcal{W}$ be our working set, which is the set of all expressions of length at most two that can be formed using $(\mathcal{C} \cup \mathcal{X})$. Also, V be the set of all program points, with a special point `START` denoting the beginning of the program.

Three global variables are maintained -

- `Partitions` : It is two-dimensional integer array indexed by the elements of sets V and $\mathcal{W}$ respectively. It helps to keep track of partitions at some $v \in V$ by holding same integer at `Partitions[`v`][`e`]` and `Partitions[`v`][`e'`]`, for $e, e' \in \mathcal{W}$ if they belongs to the same equivalence class at v. Basically, the array maintains set identifiers which helps to identify whether two expressions belong to the same sets (equivalence classes).
- `SetCnt` : It helps to keep track of the next set identifier (an integer) to be used. Whenever, a set identifier is needed the current value of `SetCnt` is used and at the same time it is incremented so that the same number is never used again.
- `Parent` : It is `map` indexed by a tuple of three elements - an operator and two set identifiers. Whenever an expression $(x + y)$ is assigned a set identifier c, c is stored in `Parent` with $\{+, a, b\}$ as key, where a and b are the set identifiers of x and y respectively. Next time when a set identifier for an expression $(x' + y')$ is needed, where x' and y' have identifiers a and b respectively, c is used instead of using a new set identifier - the `Parent` of $\{+, a, b\}$ that was earlier stored in the map.

There are a few functions whose definitions are not required explicitly -

- `OPERATOR(`e`)` : Returns operator used in expression $e \in (\mathcal{W} \setminus (\mathcal{C} \cup \mathcal{X}))$
- `LEFT(`e`)` : Returns left operand of expression $e \in (\mathcal{W} \setminus (\mathcal{C} \cup \mathcal{X}))$
- `RIGHT(`e`)` : Returns right operand of expression $e \in (\mathcal{W} \setminus (\mathcal{C} \cup \mathcal{X}))$
- `PREDECESSORS(`v`)` : Returns the set of predecessors of program point $v \in V$

NOTE - For implementation, the identifiers corresponding to $\top$ partition should be such that they don't occur in normal partitions and are also easily distinguishable from them. An easy choice for consistency is to use non-negative integers as normal set identifiers and an array of -1 to refer $\top$ partition.

4.2 Pseudocode

Algorithm 1 Main Herbrand Equivalence Analysis Function

```
procedure HerbrandAnalysis()
                                  ▷ % Initialise Partitions for all program points %
    for v ∈ V do
        Partitions[v] ← ⊤
                                  ▷ % Update Partitions for START point %
    findInitialPartition()
                                  ▷ % Process all program points till convergence %
    converged ← false
    while converged is False do
        converged ← true
        for v ∈ (V \ {START}) do
            oldPartition ← Partitions[v]
                                  ▷ % Update Partitions at v %
            if v is a Transfer Point then
                TransferFunction(v)
            else
                ConfluenceFunction(v)
                                  ▷ $ Update convergence flag %
            if not SamePartition(oldPartition, Partitions[v]) then
                converged ← false
```

Algorithm 2 Transfer Function

```
procedure TransferFunction(v : x ← e)
    u ← Predecessors(v)
    Partitions[v] ← Partitions[u]
                                  ▷ % Update set identifier for x %
    if e is Deterministic then
        Partitions[v][x] ← Partitions[v][e]
    else
        Partitions[v][x] ← SetCtr++
                                  ▷ % Update set identifiers for expressions containing x %
    for {e′ ∈ (W \ (C ∪ X)) | x ∈ e′} do
        Partitions[v][e′] ← GetSetId(v, e′)
```

Algorithm 3 Confluence Function

```
procedure ConfluenceFunction(v)
        ▷ If all predecessor partitions are ⊤ then current partition will also be ⊤
    continueFlag ← false
    for u ∈ Predecessors(v) do
        if Partitions[u] ≠ ⊤ then
            continueFlag ← true

    if continueFlag is False then
        Partitions[v] ← ⊤
        return
                                    ▷ accessFlag keeps track of processed expressions
    for e ∈ 𝒲 do
        accessFlag[e] ← false
                                    ▷ Process all expressions if they are still unprocessed
    for e ∈ 𝒲 do
        if accessFlag[e] is False then
                            ▷ PredIDs is the set of set identifiers of e at its predecessors
            PredIDs ← φ
            for u ∈ Predecessors(v) do
                if Partitions[u] ≠ ⊤ then
                    PredIDs ← (PredIDs ∪ Partitions[u][e])

            if PredIDs is Singleton then
                Partitions[v][e] ← PredIDs
                accessFlag[e] ← true
            else
                 ▷ expClass holds e′ ∈ 𝒲 that are equivalent to e at all predecessors of v
                expClass ← 𝒲
                for u ∈ Predecessors(v) do
                    if Partitions[u] ≠ ⊤ then
                        expClass ← (expClass ∩ GetClass(u, e))
                                                        ▷ Update Partitions map
                newSetID ← SetCnt++
                for e′ ∈ expClass do
                    Partitions[v][e′] ← newSetID
                    acessFlag[e′] ← true

                                                        ▷ Update Parent map
    for e ∈ (𝒲 \ (𝒞 ∪ 𝒳)) do
        op ← Operator(e)
        leftSetID ← Partitions[Left(e)]
        rightSetID ← Partitions[Right(e)]
        Parent[{op, leftSetID, rightSetID}] ← Partitions[v][e]
```

Algorithm 4 Checks whether two partitions are same or not

```
procedure SamePartition(first, second)
    for e ∈ 𝒲 do
        if GetClass(first, e) ≠ GetClass(second, e) then
            return false
    return true
```

Algorithm 5 Finds equivalence class of an expression in a partition

```
procedure GetClass(partition, e)
    expClass ← φ
    for e′ ∈ 𝒲 do
        if partition[e′] == partition[e] then
            expClass ← (expClass ∪ {e′})
    return expClass
```

Algorithm 6 Initialises partition for START point

```
procedure FindInitialPartition()
    for x ∈ (𝒞 ∪ 𝒳) do
        Partitions[START][x] ← SetCtr++
    for e ∈ (𝒲 \ (𝒞 ∪ 𝒳)) do
        Partitions[START][e] ← GetSetID(Partitions[START], e)
```

Algorithm 7 Finds and returns set identifier for a two length expression in a partition, by looking at its operands and operator

```
procedure GetSetID(partition, e)
    op ← Operator(e)
    leftSetID ← partition[Left(e)]
    rightSetID ← partition[Right(e)]

    if not defined Parent[{op, leftSetID, rightSetID}] then
        Parent[{op, leftSetID, rightSetID}] ← SetCtr++

    return Parent[{op, leftSetID, rightSetID}]
```

4.3 Updates Over Original Algorithm

Following major improvements/corrections have been made to the original algorithm mentioned in [4].

- **Representing Partitions**

  The original algorithm uses `ID structure` to maintain equivalence information. A two-dimensional array `Partitions` having an entry for each program point and each expression, stores a pointer to an `ID object`. At a program point, two expressions are equivalent if they contain pointers to the same object. Though this approach is correct, it adds a lot of overhead in the implementation both in terms of time and space - the `ID objects` have to be created and destroyed during runtime, their data fields have to be maintained properly etc.

  The updated algorithm solves this problem by using just integer set identifiers instead of pointers to some dynamically created objects and the `Parent map`. Again same as before, two expressions are equivalent at a program point if `Partitions` array stores same set identifiers for the expressions at that program point. `Parent` map captures the relation between the set identifiers. This modification makes the actual implementation to be more simpler, efficient and intuitive.

- **Confluence Function**

  The `Confluence` function in the original algorithm processes only the set $\mathcal{C} \cup \mathcal{V}$ in its main loop. This is wrong and instead the whole working set $\mathcal{W}$ should be considered. As an example, consider the test case in Section 5.15. At the confluence point, y and $x+2$ must be equivalent, but the original algorithm assigns them pointers to different `ID objects` indicating they are not equivalent.

- **Non-deterministic Assignment**

  Non-deterministic assignment need not be handled separately and a single transfer function is sufficent (see Algorithm 2).

Chapter 5

Test Cases

This chapter includes some test cases/examples on which the Herbrand analysis algorithm can be checked for correctness. They are representative examples chosen to cover different scenarios that arise during program analysis for computing Herbrand equivalences.

5.1 Test Case 1

This is a very basic example that demonstrates how transfer function modifies the partitions through the program control flow.

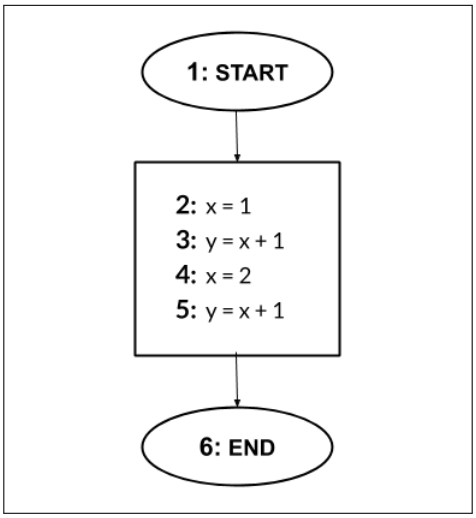

Figure 5.1 Test Case 1

- **[1] : START**

  $\{1\}$, $\{2\}$, $\{x\}$, $\{y\}$, $\{1+1\}$, $\{1+2\}$, $\{1+x\}$, $\{1+y\}$, $\{2+1\}$, $\{2+2\}$, $\{2+x\}$, $\{2+y\}$, $\{x+1\}$, $\{x+2\}$, $\{x+x\}$, $\{x+y\}$, $\{y+1\}$, $\{y+2\}$, $\{y+x\}$, $\{y+y\}$

- **[2] : Transfer Point** $\Rightarrow x = 1$

  $\{1, x\}$, $\{2\}$, $\{y\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+2, x+2\}$, $\{1+y, x+y\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{2+y\}$, $\{y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$

- **[3] : Transfer Point** $\Rightarrow y = x + 1$

  $\{1, x\}$, $\{2\}$, $\{y, 1+1, 1+x, x+1, x+x\}$, $\{1+2, x+2\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{1+y, x+y\}$, $\{2+y\}$, $\{y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$

- **[4] : Transfer Point** $\Rightarrow x = 2$

  $\{1\}$, $\{2, x\}$, $\{y, 1+1\}$, $\{1+2, 1+x\}$, $\{2+1, x+1\}$, $\{2+2, 2+x, x+2, x+x\}$, $\{1+y\}$, $\{2+y, x+y\}$, $\{y+1\}$, $\{y+2, y+x\}$, $\{y+y\}$

- **[5] : Transfer Point** $\Rightarrow y = x + 1$

  $\{1\}$, $\{2, x\}$, $\{1+1\}$, $\{1+2, 1+x\}$, $\{y, 2+1, x+1\}$, $\{2+2, 2+x, x+2, x+x\}$, $\{1+y\}$, $\{2+y, x+y\}$, $\{y+1\}$, $\{y+2, y+x\}$, $\{y+y\}$

- **[6] : END**

  $\{1\}$, $\{2, x\}$, $\{1+1\}$, $\{1+2, 1+x\}$, $\{y, 2+1, x+1\}$, $\{2+2, 2+x, x+2, x+x\}$, $\{1+y\}$, $\{2+y, x+y\}$, $\{y+1\}$, $\{y+2, y+x\}$, $\{y+y\}$

5.2 Test Case 2

This is another basic example that demonstrates non-deterministic assignment.

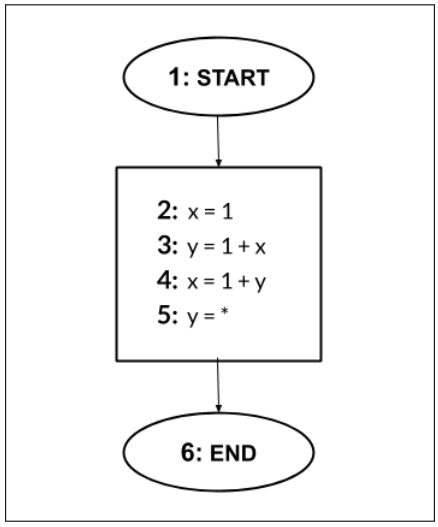

Figure 5.2 Test Case 2

- **[1] : START**

  $\{1\}$, $\{x\}$, $\{y\}$, $\{1+1\}$, $\{1+x\}$, $\{1+y\}$, $\{x+1\}$, $\{x+x\}$, $\{x+y\}$, $\{y+1\}$, $\{y+x\}$, $\{y+y\}$

- **[2] : Transfer Point** $\Rightarrow x = 1$

  $\{1, x\}$, $\{y\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+y, x+y\}$, $\{y+1, y+x\}$, $\{y+y\}$

- **[3] : Transfer Point** $\Rightarrow y = 1 + x$

  $\{1, x\}$, $\{y, 1+1, 1+x, x+1, x+x\}$, $\{1+y, x+y\}$, $\{y+1, y+x\}$, $\{y+y\}$

- **[4] : Transfer Point** $\Rightarrow x = 1 + y$

  $\{1\}$, $\{y, 1+1\}$, $\{x, 1+y\}$, $\{y+1\}$, $\{y+y\}$, $\{1+x\}$, $\{x+1\}$, $\{x+x\}$, $\{x+y\}$, $\{y+x\}$

- **[5] : Transfer Point** $\Rightarrow y = *$

  $\{1\}$, $\{1+1\}$, $\{x\}$, $\{1+x\}$, $\{x+1\}$, $\{x+x\}$, $\{y\}$, $\{1+y\}$, $\{x+y\}$, $\{y+1\}$, $\{y+x\}$, $\{y+y\}$

- **[6] : END**

  $\{1\}$, $\{1+1\}$, $\{x\}$, $\{1+x\}$, $\{x+1\}$, $\{x+x\}$, $\{y\}$, $\{1+y\}$, $\{x+y\}$, $\{y+1\}$, $\{y+x\}$, $\{y+y\}$

5.3 Test Case 3

Here at the **END** program point, y and z must be equivalent to each other but u and v must not.

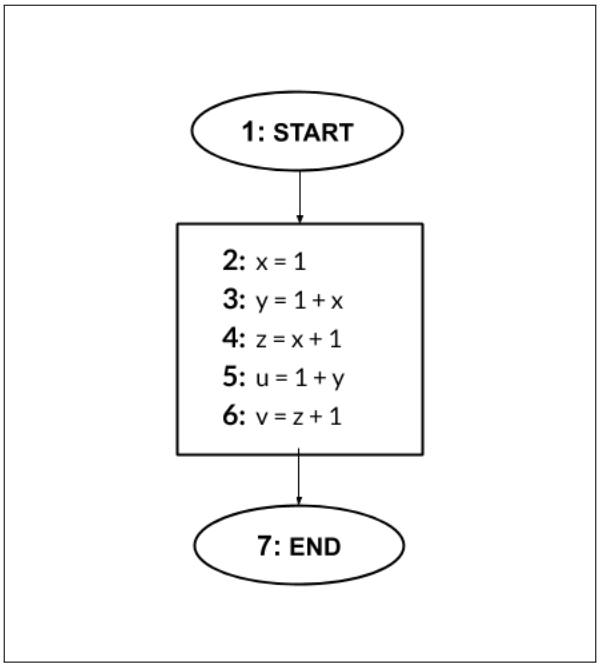

Figure 5.3 Test Case 3

- **[1] : START**

  $\{1\}$, $\{x\}$, $\{y\}$, $\{z\}$, $\{u\}$, $\{v\}$, $\{1+1\}$, $\{1+x\}$, $\{1+y\}$, $\{1+z\}$, $\{1+u\}$, $\{1+v\}$, $\{x+1\}$, $\{x+x\}$, $\{x+y\}$, $\{x+z\}$, $\{x+u\}$, $\{x+v\}$, $\{y+1\}$, $\{y+x\}$, $\{y+y\}$, $\{y+z\}$, $\{y+u\}$, $\{y+v\}$, $\{z+1\}$, $\{z+x\}$, $\{z+y\}$, $\{z+z\}$, $\{z+u\}$, $\{z+v\}$, $\{u+1\}$, $\{u+x\}$, $\{u+y\}$, $\{u+z\}$, $\{u+u\}$, $\{u+v\}$, $\{v+1\}$, $\{v+x\}$, $\{v+y\}$, $\{v+z\}$, $\{v+u\}$, $\{v+v\}$

- **[2] : Transfer Point** $\Rightarrow x = 1$

  $\{1, x\}$, $\{y\}$, $\{z\}$, $\{u\}$, $\{v\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+y, x+y\}$, $\{1+z, x+z\}$, $\{1+u, x+u\}$, $\{1+v, x+v\}$, $\{y+1, y+x\}$, $\{y+y\}$, $\{y+z\}$, $\{y+u\}$, $\{y+v\}$, $\{z+1, z+x\}$, $\{z+y\}$, $\{z+z\}$, $\{z+u\}$, $\{z+v\}$, $\{u+1, u+x\}$, $\{u+y\}$, $\{u+z\}$, $\{u+u\}$, $\{u+v\}$, $\{v+1, v+x\}$, $\{v+y\}$, $\{v+z\}$, $\{v+u\}$, $\{v+v\}$

- **[3] : Transfer Point** $\Rightarrow y = 1 + x$

$\{1, x\}$, $\{z\}$, $\{u\}$, $\{v\}$, $\{y, 1+1, 1+x, x+1, x+x\}$, $\{1+z, x+z\}$, $\{1+u, x+u\}$, $\{1+v, x+v\}$, $\{z+1, z+x\}$, $\{z+z\}$, $\{z+u\}$, $\{z+v\}$, $\{u+1, u+x\}$, $\{u+z\}$, $\{u+u\}$, $\{u+v\}$, $\{v+1, v+x\}$, $\{v+z\}$, $\{v+u\}$, $\{v+v\}$, $\{1+y, x+y\}$, $\{y+1, y+x\}$, $\{y+y\}$, $\{y+z\}$, $\{y+u\}$, $\{y+v\}$, $\{z+y\}$, $\{u+y\}$, $\{v+y\}$

- **[4] : Transfer Point** $\Rightarrow z = x + 1$

  $\{1, x\}$, $\{u\}$, $\{v\}$, $\{y, z, 1+1, 1+x, x+1, x+x\}$, $\{1+u, x+u\}$, $\{1+v, x+v\}$, $\{u+1, u+x\}$, $\{u+u\}$, $\{u+v\}$, $\{v+1, v+x\}$, $\{v+u\}$, $\{v+v\}$, $\{1+y, 1+z, x+y, x+z\}$, $\{y+1, y+x, z+1, z+x\}$, $\{y+y, y+z, z+y, z+z\}$, $\{y+u, z+u\}$, $\{y+v, z+v\}$, $\{u+y, u+z\}$, $\{v+y, v+z\}$

- **[5] : Transfer Point** $\Rightarrow u = 1 + y$

  $\{1, x\}$, $\{v\}$, $\{y, z, 1+1, 1+x, x+1, x+x\}$, $\{1+v, x+v\}$, $\{v+1, v+x\}$, $\{v+v\}$, $\{u, 1+y, 1+z, x+y, x+z\}$, $\{y+1, y+x, z+1, z+x\}$, $\{y+y, y+z, z+y, z+z\}$, $\{y+v, z+v\}$, $\{v+y, v+z\}$, $\{1+u, x+u\}$, $\{y+u, z+u\}$, $\{u+1, u+x\}$, $\{u+y, u+z\}$, $\{u+u\}$, $\{u+v\}$, $\{v+u\}$

- **[6] : Transfer Point** $\Rightarrow v = z + 1$

  $\{1, x\}$, $\{y, z, 1+1, 1+x, x+1, x+x\}$, $\{u, 1+y, 1+z, x+y, x+z\}$, $\{v, y+1, y+x, z+1, z+x\}$, $\{y+y, y+z, z+y, z+z\}$, $\{1+u, x+u\}$, $\{y+u, z+u\}$, $\{u+1, u+x\}$, $\{u+y, u+z\}$, $\{u+u\}$, $\{1+v, x+v\}$, $\{y+v, z+v\}$, $\{u+v\}$, $\{v+1, v+x\}$, $\{v+y, v+z\}$, $\{v+u\}$, $\{v+v\}$

- **[7] : END**

  $\{1, x\}$, $\{y, z, 1+1, 1+x, x+1, x+x\}$, $\{u, 1+y, 1+z, x+y, x+z\}$, $\{v, y+1, y+x, z+1, z+x\}$, $\{y+y, y+z, z+y, z+z\}$, $\{1+u, x+u\}$, $\{y+u, z+u\}$, $\{u+1, u+x\}$, $\{u+y, u+z\}$, $\{u+u\}$, $\{1+v, x+v\}$, $\{y+v, z+v\}$, $\{u+v\}$, $\{v+1, v+x\}$, $\{v+y, v+z\}$, $\{v+u\}$, $\{v+v\}$

5.4 Test Case 4

Here at the **END** program point, t and d must be equivalent.

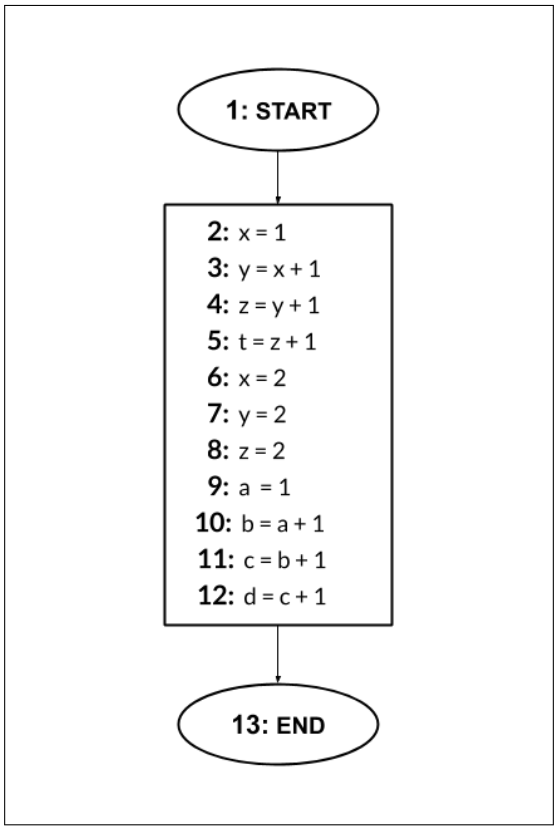

Figure 5.4 Test Case 4

- **[1] : START**

  $\{1\}$, $\{2\}$, $\{x\}$, $\{y\}$, $\{z\}$, $\{t\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{1+1\}$, $\{1+2\}$, $\{1+x\}$, $\{1+y\}$, $\{1+z\}$, $\{1+t\}$, $\{1+a\}$, $\{1+b\}$, $\{1+c\}$, $\{1+d\}$, $\{2+1\}$, $\{2+2\}$, $\{2+x\}$, $\{2+y\}$, $\{2+z\}$, $\{2+t\}$, $\{2+a\}$, $\{2+b\}$, $\{2+c\}$, $\{2+d\}$, $\{x+1\}$, $\{x+2\}$, $\{x+x\}$, $\{x+y\}$, $\{x+z\}$, $\{x+t\}$, $\{x+a\}$, $\{x+b\}$, $\{x+c\}$, $\{x+d\}$, $\{y+1\}$, $\{y+2\}$, $\{y+x\}$, $\{y+y\}$, $\{y+z\}$, $\{y+t\}$, $\{y+a\}$, $\{y+b\}$, $\{y+c\}$, $\{y+d\}$, $\{z+1\}$, $\{z+2\}$, $\{z+x\}$, $\{z+y\}$, $\{z+z\}$, $\{z+t\}$, $\{z+a\}$, $\{z+b\}$, $\{z+c\}$, $\{z+d\}$, $\{t+1\}$, $\{t+2\}$, $\{t+x\}$, $\{t+y\}$, $\{t+z\}$, $\{t+t\}$, $\{t+a\}$, $\{t+b\}$, $\{t+c\}$, $\{t+d\}$, $\{a+1\}$, $\{a+2\}$, $\{a+x\}$, $\{a+y\}$, $\{a+z\}$, $\{a+t\}$, $\{a+a\}$, $\{a+b\}$, $\{a+c\}$, $\{a+d\}$, $\{b+1\}$, $\{b+2\}$, $\{b+x\}$, $\{b+y\}$, $\{b+z\}$, $\{b+t\}$, $\{b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+1\}$, $\{c+2\}$, $\{c+x\}$, $\{c+y\}$,

$\{c+z\}$, $\{c+t\}$, $\{c+a\}$, $\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+1\}$, $\{d+2\}$, $\{d+x\}$, $\{d+y\}$, $\{d+z\}$, $\{d+t\}$, $\{d+a\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$

- **[2] : Transfer Point** $\Rightarrow x = 1$

  $\{1, x\}$, $\{2\}$, $\{y\}$, $\{z\}$, $\{t\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+2, x+2\}$, $\{1+y, x+y\}$, $\{1+z, x+z\}$, $\{1+t, x+t\}$, $\{1+a, x+a\}$, $\{1+b, x+b\}$, $\{1+c, x+c\}$, $\{1+d, x+d\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{2+y\}$, $\{2+z\}$, $\{2+t\}$, $\{2+a\}$, $\{2+b\}$, $\{2+c\}$, $\{2+d\}$, $\{y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$, $\{y+z\}$, $\{y+t\}$, $\{y+a\}$, $\{y+b\}$, $\{y+c\}$, $\{y+d\}$, $\{z+1, z+x\}$, $\{z+2\}$, $\{z+y\}$, $\{z+z\}$, $\{z+t\}$, $\{z+a\}$, $\{z+b\}$, $\{z+c\}$, $\{z+d\}$, $\{t+1, t+x\}$, $\{t+2\}$, $\{t+y\}$, $\{t+z\}$, $\{t+t\}$, $\{t+a\}$, $\{t+b\}$, $\{t+c\}$, $\{t+d\}$, $\{a+1, a+x\}$, $\{a+2\}$, $\{a+y\}$, $\{a+z\}$, $\{a+t\}$, $\{a+a\}$, $\{a+b\}$, $\{a+c\}$, $\{a+d\}$, $\{b+1, b+x\}$, $\{b+2\}$, $\{b+y\}$, $\{b+z\}$, $\{b+t\}$, $\{b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+1, c+x\}$, $\{c+2\}$, $\{c+y\}$, $\{c+z\}$, $\{c+t\}$, $\{c+a\}$, $\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+1, d+x\}$, $\{d+2\}$, $\{d+y\}$, $\{d+z\}$, $\{d+t\}$, $\{d+a\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$

- **[3] : Transfer Point** $\Rightarrow y = x + 1$

  $\{1, x\}$, $\{2\}$, $\{z\}$, $\{t\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{y, 1+1, 1+x, x+1, x+x\}$, $\{1+2, x+2\}$, $\{1+z, x+z\}$, $\{1+t, x+t\}$, $\{1+a, x+a\}$, $\{1+b, x+b\}$, $\{1+c, x+c\}$, $\{1+d, x+d\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{2+z\}$, $\{2+t\}$, $\{2+a\}$, $\{2+b\}$, $\{2+c\}$, $\{2+d\}$, $\{z+1, z+x\}$, $\{z+2\}$, $\{z+z\}$, $\{z+t\}$, $\{z+a\}$, $\{z+b\}$, $\{z+c\}$, $\{z+d\}$, $\{t+1, t+x\}$, $\{t+2\}$, $\{t+z\}$, $\{t+t\}$, $\{t+a\}$, $\{t+b\}$, $\{t+c\}$, $\{t+d\}$, $\{a+1, a+x\}$, $\{a+2\}$, $\{a+z\}$, $\{a+t\}$, $\{a+a\}$, $\{a+b\}$, $\{a+c\}$, $\{a+d\}$, $\{b+1, b+x\}$, $\{b+2\}$, $\{b+z\}$, $\{b+t\}$, $\{b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+1, c+x\}$, $\{c+2\}$, $\{c+z\}$, $\{c+t\}$, $\{c+a\}$, $\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+1, d+x\}$, $\{d+2\}$, $\{d+z\}$, $\{d+t\}$, $\{d+a\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$, $\{1+y, x+y\}$, $\{2+y\}$, $\{y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$, $\{y+z\}$, $\{y+t\}$, $\{y+a\}$, $\{y+b\}$, $\{y+c\}$, $\{y+d\}$, $\{z+y\}$, $\{t+y\}$, $\{a+y\}$, $\{b+y\}$, $\{c+y\}$, $\{d+y\}$

- **[4] : Transfer Point** $\Rightarrow z = y + 1$

  $\{1, x\}$, $\{2\}$, $\{t\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{y, 1+1, 1+x, x+1, x+x\}$, $\{1+2, x+2\}$, $\{1+t, x+t\}$, $\{1+a, x+a\}$, $\{1+b, x+b\}$, $\{1+c, x+c\}$, $\{1+d, x+d\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{2+t\}$, $\{2+a\}$, $\{2+b\}$, $\{2+c\}$, $\{2+d\}$, $\{t+1, t+x\}$, $\{t+2\}$, $\{t+t\}$, $\{t+a\}$, $\{t+b\}$, $\{t+c\}$, $\{t+d\}$, $\{a+1, a+x\}$, $\{a+2\}$, $\{a+t\}$, $\{a+a\}$, $\{a+b\}$, $\{a+c\}$, $\{a+d\}$, $\{b+1, b+x\}$, $\{b+2\}$, $\{b+t\}$, $\{b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+1, c+x\}$, $\{c+2\}$, $\{c+t\}$, $\{c+a\}$, $\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+1, d+x\}$, $\{d+2\}$, $\{d+t\}$, $\{d+a\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$, $\{1+y, x+y\}$, $\{2+y\}$, $\{z, y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$, $\{y+t\}$, $\{y+a\}$, $\{y+b\}$, $\{y+c\}$, $\{y+d\}$, $\{t+y\}$, $\{a+y\}$, $\{b+y\}$, $\{c+y\}$, $\{d+y\}$, $\{1+z, x+z\}$, $\{2+z\}$, $\{y+z\}$, $\{z+1, z+x\}$, $\{z+2\}$, $\{z+y\}$, $\{z+z\}$, $\{z+t\}$, $\{z+a\}$, $\{z+b\}$, $\{z+c\}$, $\{z+d\}$, $\{t+z\}$, $\{a+z\}$, $\{b+z\}$, $\{c+z\}$, $\{d+z\}$

- **[5] : Transfer Point** $\Rightarrow t = z + 1$

  $\{1, x\}$, $\{2\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{y, 1+1, 1+x, x+1, x+x\}$, $\{1+2, x+2\}$, $\{1+a, x+a\}$, $\{1+b, x+b\}$, $\{1+c, x+c\}$, $\{1+d, x+d\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{2+a\}$, $\{2+b\}$, $\{2+c\}$, $\{2+d\}$, $\{a+1, a+x\}$, $\{a+2\}$, $\{a+a\}$, $\{a+b\}$, $\{a+c\}$, $\{a+d\}$, $\{b+1, b+x\}$, $\{b+2\}$, $\{b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+1, c+x\}$, $\{c+2\}$, $\{c+a\}$, $\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+1, d+x\}$, $\{d+2\}$, $\{d+a\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$, $\{1+y, x+y\}$, $\{2+y\}$, $\{z, y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$, $\{y+a\}$, $\{y+b\}$, $\{y+c\}$, $\{y+d\}$, $\{a+y\}$, $\{b+y\}$, $\{c+y\}$, $\{d+y\}$, $\{1+z, x+z\}$, $\{2+z\}$, $\{y+z\}$, $\{t, z+1, z+x\}$, $\{z+2\}$, $\{z+y\}$, $\{z+z\}$, $\{z+a\}$, $\{z+b\}$, $\{z+c\}$, $\{z+d\}$, $\{a+z\}$, $\{b+z\}$, $\{c+z\}$, $\{d+z\}$, $\{1+t, x+t\}$, $\{2+t\}$, $\{y+t\}$, $\{z+t\}$, $\{t+1, t+x\}$, $\{t+2\}$, $\{t+y\}$, $\{t+z\}$, $\{t+t\}$, $\{t+a\}$, $\{t+b\}$, $\{t+c\}$, $\{t+d\}$, $\{a+t\}$, $\{b+t\}$, $\{c+t\}$, $\{d+t\}$

- **[6] : Transfer Point** $\Rightarrow x = 2$

  $\{1\}$, $\{2, x\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{y, 1+1\}$, $\{1+2, 1+x\}$, $\{1+a\}$, $\{1+b\}$, $\{1+c\}$, $\{1+d\}$, $\{2+1, x+1\}$, $\{2+2, 2+x, x+2, x+x\}$, $\{2+a, x+a\}$, $\{2+b, x+b\}$,

$\{2+c, x+c\}$, $\{2+d, x+d\}$, $\{a+1\}$, $\{a+2, a+x\}$, $\{a+a\}$, $\{a+b\}$, $\{a+c\}$, $\{a+d\}$, $\{b+1\}$, $\{b+2, b+x\}$, $\{b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+1\}$, $\{c+2, c+x\}$, $\{c+a\}$, $\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+1\}$, $\{d+2, d+x\}$, $\{d+a\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$, $\{1+y\}$, $\{2+y, x+y\}$, $\{z, y+1\}$, $\{y+2, y+x\}$, $\{y+y\}$, $\{y+a\}$, $\{y+b\}$, $\{y+c\}$, $\{y+d\}$, $\{a+y\}$, $\{b+y\}$, $\{c+y\}$, $\{d+y\}$, $\{1+z\}$, $\{2+z, x+z\}$, $\{y+z\}$, $\{t, z+1\}$, $\{z+2, z+x\}$, $\{z+y\}$, $\{z+z\}$, $\{z+a\}$, $\{z+b\}$, $\{z+c\}$, $\{z+d\}$, $\{a+z\}$, $\{b+z\}$, $\{c+z\}$, $\{d+z\}$, $\{1+t\}$, $\{2+t, x+t\}$, $\{y+t\}$, $\{z+t\}$, $\{t+1\}$, $\{t+2, t+x\}$, $\{t+y\}$, $\{t+z\}$, $\{t+t\}$, $\{t+a\}$, $\{t+b\}$, $\{t+c\}$, $\{t+d\}$, $\{a+t\}$, $\{b+t\}$, $\{c+t\}$, $\{d+t\}$

- **[7] : Transfer Point** $\Rightarrow y = 2$

  $\{1\}$, $\{2, x, y\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{1+1\}$, $\{1+2, 1+x, 1+y\}$, $\{1+a\}$, $\{1+b\}$, $\{1+c\}$, $\{1+d\}$, $\{2+1, x+1, y+1\}$, $\{2+2, 2+x, 2+y, x+2, x+x, x+y, y+2, y+x, y+y\}$, $\{2+a, x+a, y+a\}$, $\{2+b, x+b, y+b\}$, $\{2+c, x+c, y+c\}$, $\{2+d, x+d, y+d\}$, $\{a+1\}$, $\{a+2, a+x, a+y\}$, $\{a+a\}$, $\{a+b\}$, $\{a+c\}$, $\{a+d\}$, $\{b+1\}$, $\{b+2, b+x, b+y\}$, $\{b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+1\}$, $\{c+2, c+x, c+y\}$, $\{c+a\}$, $\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+1\}$, $\{d+2, d+x, d+y\}$, $\{d+a\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$, $\{z\}$, $\{1+z\}$, $\{2+z, x+z, y+z\}$, $\{t, z+1\}$, $\{z+2, z+x, z+y\}$, $\{z+z\}$, $\{z+a\}$, $\{z+b\}$, $\{z+c\}$, $\{z+d\}$, $\{a+z\}$, $\{b+z\}$, $\{c+z\}$, $\{d+z\}$, $\{1+t\}$, $\{2+t, x+t, y+t\}$, $\{z+t\}$, $\{t+1\}$, $\{t+2, t+x, t+y\}$, $\{t+z\}$, $\{t+t\}$, $\{t+a\}$, $\{t+b\}$, $\{t+c\}$, $\{t+d\}$, $\{a+t\}$, $\{b+t\}$, $\{c+t\}$, $\{d+t\}$

- **[8] : Transfer Point** $\Rightarrow z = 2$

  $\{1\}$, $\{2, x, y, z\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{1+1\}$, $\{1+2, 1+x, 1+y, 1+z\}$, $\{1+a\}$, $\{1+b\}$, $\{1+c\}$, $\{1+d\}$, $\{2+1, x+1, y+1, z+1\}$, $\{2+2, 2+x, 2+y, 2+z, x+2, x+x, x+y, x+z, y+2, y+x, y+y, y+z, z+2, z+x, z+y, z+z\}$, $\{2+a, x+a, y+a, z+a\}$, $\{2+b, x+b, y+b, z+b\}$, $\{2+c, x+c, y+c, z+c\}$, $\{2+d, x+d, y+d, z+d\}$, $\{a+1\}$, $\{a+2, a+x, a+y, a+z\}$, $\{a+a\}$, $\{a+b\}$, $\{a+c\}$, $\{a+d\}$, $\{b+1\}$, $\{b+2, b+x, b+y, b+z\}$, $\{b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+1\}$, $\{c+2, c+x, c+y, c+z\}$, $\{c+a\}$, $\{c+b\}$,

$\{c+c\}$, $\{c+d\}$, $\{d+1\}$, $\{d+2, d+x, d+y, d+z\}$, $\{d+a\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$, $\{t\}$, $\{1+t\}$, $\{2+t, x+t, y+t, z+t\}$, $\{t+1\}$, $\{t+2, t+x, t+y, t+z\}$, $\{t+t\}$, $\{t+a\}$, $\{t+b\}$, $\{t+c\}$, $\{t+d\}$, $\{a+t\}$, $\{b+t\}$, $\{c+t\}$, $\{d+t\}$

- **[9] : Transfer Point** $\Rightarrow a = 1$

  $\{1, a\}$, $\{2, x, y, z\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{1+1, 1+a, a+1, a+a\}$, $\{1+2, 1+x, 1+y, 1+z, a+2, a+x, a+y, a+z\}$, $\{1+b, a+b\}$, $\{1+c, a+c\}$, $\{1+d, a+d\}$, $\{2+1, 2+a, x+1, x+a, y+1, y+a, z+1, z+a\}$, $\{2+2, 2+x, 2+y, 2+z, x+2, x+x, x+y, x+z, y+2, y+x, y+y, y+z, z+2, z+x, z+y, z+z\}$, $\{2+b, x+b, y+b, z+b\}$, $\{2+c, x+c, y+c, z+c\}$, $\{2+d, x+d, y+d, z+d\}$, $\{b+1, b+a\}$, $\{b+2, b+x, b+y, b+z\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+1, c+a\}$, $\{c+2, c+x, c+y, c+z\}$, $\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+1, d+a\}$, $\{d+2, d+x, d+y, d+z\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$, $\{t\}$, $\{1+t, a+t\}$, $\{2+t, x+t, y+t, z+t\}$, $\{t+1, t+a\}$, $\{t+2, t+x, t+y, t+z\}$, $\{t+t\}$, $\{t+b\}$, $\{t+c\}$, $\{t+d\}$, $\{b+t\}$, $\{c+t\}$, $\{d+t\}$

- **[10] : Transfer Point** $\Rightarrow b = a + 1$

  $\{1, a\}$, $\{2, x, y, z\}$, $\{c\}$, $\{d\}$, $\{b, 1+1, 1+a, a+1, a+a\}$, $\{1+2, 1+x, 1+y, 1+z, a+2, a+x, a+y, a+z\}$, $\{1+c, a+c\}$, $\{1+d, a+d\}$, $\{2+1, 2+a, x+1, x+a, y+1, y+a, z+1, z+a\}$, $\{2+2, 2+x, 2+y, 2+z, x+2, x+x, x+y, x+z, y+2, y+x, y+y, y+z, z+2, z+x, z+y, z+z\}$, $\{2+c, x+c, y+c, z+c\}$, $\{2+d, x+d, y+d, z+d\}$, $\{c+1, c+a\}$, $\{c+2, c+x, c+y, c+z\}$, $\{c+c\}$, $\{c+d\}$, $\{d+1, d+a\}$, $\{d+2, d+x, d+y, d+z\}$, $\{d+c\}$, $\{d+d\}$, $\{1+b, a+b\}$, $\{2+b, x+b, y+b, z+b\}$, $\{b+1, b+a\}$, $\{b+2, b+x, b+y, b+z\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+b\}$, $\{d+b\}$, $\{t\}$, $\{1+t, a+t\}$, $\{2+t, x+t, y+t, z+t\}$, $\{b+t\}$, $\{t+1, t+a\}$, $\{t+2, t+x, t+y, t+z\}$, $\{t+b\}$, $\{t+t\}$, $\{t+c\}$, $\{t+d\}$, $\{c+t\}$, $\{d+t\}$

- **[11] : Transfer Point** $\Rightarrow c = b + 1$

  $\{1, a\}$, $\{2, x, y, z\}$, $\{d\}$, $\{b, 1+1, 1+a, a+1, a+a\}$, $\{1+2, 1+x, 1+y, 1+z, a+2, a+x, a+y, a+z\}$, $\{1+d, a+d\}$, $\{2+1, 2+a, x+1, x+a, y+1, y+a, z+1, z+a\}$, $\{2+2, 2+$

$x, 2+y, 2+z, x+2, x+x, x+y, x+z, y+2, y+x, y+y, y+z, z+2, z+x, z+y, z+z\}$,
$\{2+d, x+d, y+d, z+d\}$, $\{d+1, d+a\}$, $\{d+2, d+x, d+y, d+z\}$, $\{d+d\}$, $\{1+b, a+b\}$,
$\{2+b, x+b, y+b, z+b\}$, $\{c, b+1, b+a\}$, $\{b+2, b+x, b+y, b+z\}$, $\{b+b\}$, $\{b+d\}$, $\{d+b\}$,
$\{1+c, a+c\}$, $\{2+c, x+c, y+c, z+c\}$, $\{b+c\}$, $\{t, c+1, c+a\}$, $\{c+2, c+x, c+y, c+z\}$,
$\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+c\}$, $\{1+t, a+t\}$, $\{2+t, x+t, y+t, z+t\}$, $\{b+t\}$,
$\{c+t\}$, $\{t+1, t+a\}$, $\{t+2, t+x, t+y, t+z\}$, $\{t+b\}$, $\{t+c\}$, $\{t+t\}$, $\{t+d\}$, $\{d+t\}$

- **[12] : Transfer Point** $\Rightarrow d = c + 1$

  $\{1, a\}$, $\{2, x, y, z\}$, $\{b, 1+1, 1+a, a+1, a+a\}$, $\{1+2, 1+x, 1+y, 1+z, a+2, a+x, a+$
  $y, a+z\}$, $\{2+1, 2+a, x+1, x+a, y+1, y+a, z+1, z+a\}$, $\{2+2, 2+x, 2+y, 2+z, x+$
  $2, x+x, x+y, x+z, y+2, y+x, y+y, y+z, z+2, z+x, z+y, z+z\}$, $\{1+b, a+b\}$,
  $\{2+b, x+b, y+b, z+b\}$, $\{c, b+1, b+a\}$, $\{b+2, b+x, b+y, b+z\}$, $\{b+b\}$, $\{1+c, a+c\}$,
  $\{2+c, x+c, y+c, z+c\}$, $\{b+c\}$, $\{t, d, c+1, c+a\}$, $\{c+2, c+x, c+y, c+z\}$, $\{c+b\}$, $\{c+c\}$,
  $\{1+t, 1+d, a+t, a+d\}$, $\{2+t, 2+d, x+t, x+d, y+t, y+d, z+t, z+d\}$, $\{b+t, b+d\}$,
  $\{c+t, c+d\}$, $\{t+1, t+a, d+1, d+a\}$, $\{t+2, t+x, t+y, t+z, d+2, d+x, d+y, d+z\}$,
  $\{t+b, d+b\}$, $\{t+c, d+c\}$, $\{t+t, t+d, d+t, d+d\}$,

- **[13] : END**

  $\{1, a\}$, $\{2, x, y, z\}$, $\{b, 1+1, 1+a, a+1, a+a\}$, $\{1+2, 1+x, 1+y, 1+z, a+2, a+x, a+$
  $y, a+z\}$, $\{2+1, 2+a, x+1, x+a, y+1, y+a, z+1, z+a\}$, $\{2+2, 2+x, 2+y, 2+z, x+$
  $2, x+x, x+y, x+z, y+2, y+x, y+y, y+z, z+2, z+x, z+y, z+z\}$, $\{1+b, a+b\}$,
  $\{2+b, x+b, y+b, z+b\}$, $\{c, b+1, b+a\}$, $\{b+2, b+x, b+y, b+z\}$, $\{b+b\}$, $\{1+c, a+c\}$,
  $\{2+c, x+c, y+c, z+c\}$, $\{b+c\}$, $\{t, d, c+1, c+a\}$, $\{c+2, c+x, c+y, c+z\}$, $\{c+b\}$, $\{c+c\}$,
  $\{1+t, 1+d, a+t, a+d\}$, $\{2+t, 2+d, x+t, x+d, y+t, y+d, z+t, z+d\}$, $\{b+t, b+d\}$,
  $\{c+t, c+d\}$, $\{t+1, t+a, d+1, d+a\}$, $\{t+2, t+x, t+y, t+z, d+2, d+x, d+y, d+z\}$,
  $\{t+b, d+b\}$, $\{t+c, d+c\}$, $\{t+t, t+d, d+t, d+d\}$

5.5 Test Case 5

Here at the **END** program point, y and 1 must be equivalent but x must not be equivalent to either of 1 and 2.

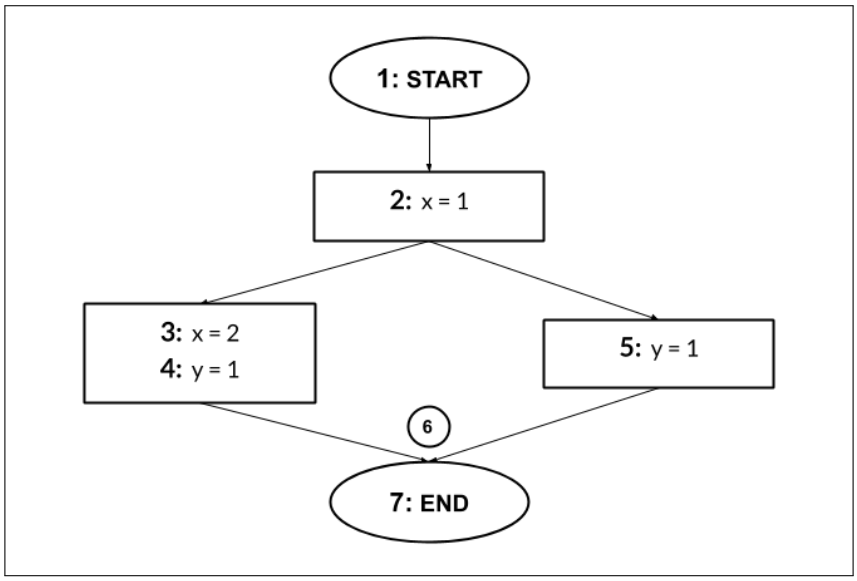

Figure 5.5 Test Case 5

- **[1] : START**

  $\{1\}$, $\{2\}$, $\{x\}$, $\{y\}$, $\{1+1\}$, $\{1+2\}$, $\{1+x\}$, $\{1+y\}$, $\{2+1\}$, $\{2+2\}$, $\{2+x\}$, $\{2+y\}$, $\{x+1\}$, $\{x+2\}$, $\{x+x\}$, $\{x+y\}$, $\{y+1\}$, $\{y+2\}$, $\{y+x\}$, $\{y+y\}$

- **[2] : Transfer Point** $\Rightarrow x = 1$

  $\{1, x\}$, $\{2\}$, $\{y\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+2, x+2\}$, $\{1+y, x+y\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{2+y\}$, $\{y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$

- **[3] : Transfer Point** $\Rightarrow x = 2$

  $\{1\}$, $\{2, x\}$, $\{y\}$, $\{1+1\}$, $\{1+2, 1+x\}$, $\{1+y\}$, $\{2+1, x+1\}$, $\{2+2, 2+x, x+2, x+x\}$, $\{2+y, x+y\}$, $\{y+1\}$, $\{y+2, y+x\}$, $\{y+y\}$

- **[4] : Transfer Point** $\Rightarrow y = 1$

  $\{1, y\}$, $\{2, x\}$, $\{1+1, 1+y, y+1, y+y\}$, $\{1+2, 1+x, y+2, y+x\}$, $\{2+1, 2+y, x+1, x+y\}$, $\{2+2, 2+x, x+2, x+x\}$

- **[5] : Transfer Point** $\Rightarrow y = 1$

  $\{1, x, y\}$, $\{2\}$, $\{1{+}1, 1{+}x, 1{+}y, x{+}1, x{+}x, x{+}y, y{+}1, y{+}x, y{+}y\}$, $\{1{+}2, x{+}2, y{+}2\}$, $\{2+1, 2+x, 2+y\}$, $\{2+2\}$

- **[6] : Confluence of** $\{4, 5\}$

  $\{1, y\}$, $\{2\}$, $\{1+1, 1+y, y+1, y+y\}$, $\{1+2, y+2\}$, $\{2+1, 2+y\}$, $\{2+2\}$, $\{x\}$, $\{1+x, y+x\}$, $\{2+x\}$, $\{x+1, x+y\}$, $\{x+2\}$, $\{x+x\}$

- **[7] : END**

  $\{1, y\}$, $\{2\}$, $\{1+1, 1+y, y+1, y+y\}$, $\{1+2, y+2\}$, $\{2+1, 2+y\}$, $\{2+2\}$, $\{x\}$, $\{1+x, y+x\}$, $\{2+x\}$, $\{x+1, x+y\}$, $\{x+2\}$, $\{x+x\}$

5.6 Test Case 6

Here at the **END** program point, a, b, c and d must be mutually non-equivalent and d must be equivalent to 1.

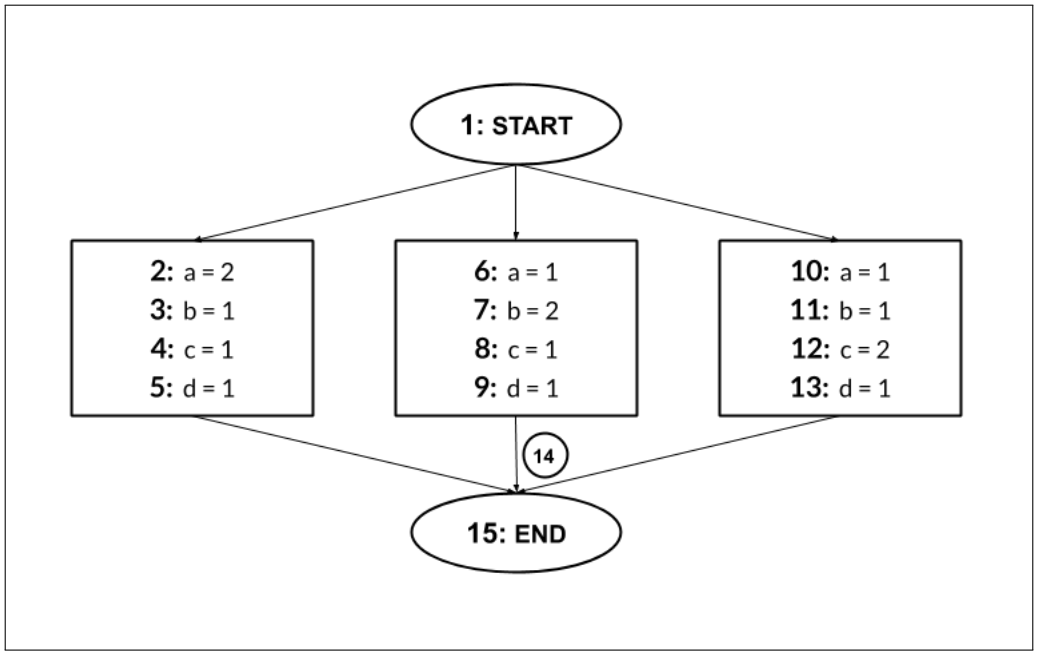

Figure 5.6 Test Case 6

- **[1] : START**

  $\{2\}$, $\{1\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{2+2\}$, $\{2+1\}$, $\{2+a\}$, $\{2+b\}$, $\{2+c\}$, $\{2+d\}$, $\{1+2\}$, $\{1+1\}$, $\{1+a\}$, $\{1+b\}$, $\{1+c\}$, $\{1+d\}$, $\{a+2\}$, $\{a+1\}$, $\{a+a\}$, $\{a+b\}$, $\{a+c\}$, $\{a+d\}$, $\{b+2\}$, $\{b+1\}$, $\{b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+2\}$, $\{c+1\}$, $\{c+a\}$, $\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+2\}$, $\{d+1\}$, $\{d+a\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$

- **[2] : Transfer Point** $\Rightarrow a = 2$

  $\{2, a\}$, $\{1\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{2+2, 2+a, a+2, a+a\}$, $\{2+1, a+1\}$, $\{2+b, a+b\}$, $\{2+c, a+c\}$, $\{2+d, a+d\}$, $\{1+2, 1+a\}$, $\{1+1\}$, $\{1+b\}$, $\{1+c\}$, $\{1+d\}$, $\{b+2, b+a\}$, $\{b+1\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+2, c+a\}$, $\{c+1\}$, $\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+2, d+a\}$, $\{d+1\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$

- **[3] : Transfer Point** $\Rightarrow b = 1$

  $\{2, a\}$, $\{1, b\}$, $\{c\}$, $\{d\}$, $\{2+2, 2+a, a+2, a+a\}$, $\{2+1, 2+b, a+1, a+b\}$, $\{2+c, a+c\}$, $\{2+d, a+d\}$, $\{1+2, 1+a, b+2, b+a\}$, $\{1+1, 1+b, b+1, b+b\}$, $\{1+c, b+c\}$, $\{1+d, b+d\}$, $\{c+2, c+a\}$, $\{c+1, c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+2, d+a\}$, $\{d+1, d+b\}$, $\{d+c\}$, $\{d+d\}$

- **[4] : Transfer Point** $\Rightarrow c = 1$

  $\{2, a\}$, $\{1, b, c\}$, $\{d\}$, $\{2+2, 2+a, a+2, a+a\}$, $\{2+1, 2+b, 2+c, a+1, a+b, a+c\}$, $\{2+d, a+d\}$, $\{1+2, 1+a, b+2, b+a, c+2, c+a\}$, $\{1+1, 1+b, 1+c, b+1, b+b, b+c, c+1, c+b, c+c\}$, $\{1+d, b+d, c+d\}$, $\{d+2, d+a\}$, $\{d+1, d+b, d+c\}$, $\{d+d\}$

- **[5] : Transfer Point** $\Rightarrow d = 1$

  $\{2, a\}$, $\{1, b, c, d\}$, $\{2+2, 2+a, a+2, a+a\}$, $\{2+1, 2+b, 2+c, 2+d, a+1, a+b, a+c, a+d\}$, $\{1+2, 1+a, b+2, b+a, c+2, c+a, d+2, d+a\}$, $\{1+1, 1+b, 1+c, 1+d, b+1, b+b, b+c, b+d, c+1, c+b, c+c, c+d, d+1, d+b, d+c, d+d\}$

- **[6] : Transfer Point** $\Rightarrow a = 1$

  $\{2\}$, $\{1, a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{2+2\}$, $\{2+1, 2+a\}$, $\{2+b\}$, $\{2+c\}$, $\{2+d\}$, $\{1+2, a+2\}$,

$\{1+1, 1+a, a+1, a+a\}$, $\{1+b, a+b\}$, $\{1+c, a+c\}$, $\{1+d, a+d\}$, $\{b+2\}$, $\{b+1, b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+2\}$, $\{c+1, c+a\}$, $\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+2\}$, $\{d+1, d+a\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$

- **[7] : Transfer Point** $\Rightarrow b = 2$

  $\{2, b\}$, $\{1, a\}$, $\{c\}$, $\{d\}$, $\{2+2, 2+b, b+2, b+b\}$, $\{2+1, 2+a, b+1, b+a\}$, $\{2+c, b+c\}$, $\{2+d, b+d\}$, $\{1+2, 1+b, a+2, a+b\}$, $\{1+1, 1+a, a+1, a+a\}$, $\{1+c, a+c\}$, $\{1+d, a+d\}$, $\{c+2, c+b\}$, $\{c+1, c+a\}$, $\{c+c\}$, $\{c+d\}$, $\{d+2, d+b\}$, $\{d+1, d+a\}$, $\{d+c\}$, $\{d+d\}$

- **[8] : Transfer Point** $\Rightarrow c = 1$

  $\{2, b\}$, $\{1, a, c\}$, $\{d\}$, $\{2+2, 2+b, b+2, b+b\}$, $\{2+1, 2+a, 2+c, b+1, b+a, b+c\}$, $\{2+d, b+d\}$, $\{1+2, 1+b, a+2, a+b, c+2, c+b\}$, $\{1+1, 1+a, 1+c, a+1, a+a, a+c, c+1, c+a, c+c\}$, $\{1+d, a+d, c+d\}$, $\{d+2, d+b\}$, $\{d+1, d+a, d+c\}$, $\{d+d\}$

- **[9] : Transfer Point** $\Rightarrow d = 1$

  $\{2, b\}$, $\{1, a, c, d\}$, $\{2+2, 2+b, b+2, b+b\}$, $\{2+1, 2+a, 2+c, 2+d, b+1, b+a, b+c, b+d\}$, $\{1+2, 1+b, a+2, a+b, c+2, c+b, d+2, d+b\}$, $\{1+1, 1+a, 1+c, 1+d, a+1, a+a, a+c, a+d, c+1, c+a, c+c, c+d, d+1, d+a, d+c, d+d\}$

- **[10] : Transfer Point** $\Rightarrow a = 1$

  $\{2\}$, $\{1, a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{2+2\}$, $\{2+1, 2+a\}$, $\{2+b\}$, $\{2+c\}$, $\{2+d\}$, $\{1+2, a+2\}$, $\{1+1, 1+a, a+1, a+a\}$, $\{1+b, a+b\}$, $\{1+c, a+c\}$, $\{1+d, a+d\}$, $\{b+2\}$, $\{b+1, b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{c+2\}$, $\{c+1, c+a\}$, $\{c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+2\}$, $\{d+1, d+a\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$

- **[11] : Transfer Point** $\Rightarrow b = 1$

  $\{2\}$, $\{1, a, b\}$, $\{c\}$, $\{d\}$, $\{2+2\}$, $\{2+1, 2+a, 2+b\}$, $\{2+c\}$, $\{2+d\}$, $\{1+2, a+2, b+2\}$, $\{1+1, 1+a, 1+b, a+1, a+a, a+b, b+1, b+a, b+b\}$, $\{1+c, a+c, b+c\}$, $\{1+d, a+d, b+d\}$, $\{c+2\}$, $\{c+1, c+a, c+b\}$, $\{c+c\}$, $\{c+d\}$, $\{d+2\}$, $\{d+1, d+a, d+b\}$, $\{d+c\}$,

$\{d+d\}$

- **[12] : Transfer Point** $\Rightarrow c = 2$

  $\{2, c\}$, $\{1, a, b\}$, $\{d\}$, $\{2+2, 2+c, c+2, c+c\}$, $\{2+1, 2+a, 2+b, c+1, c+a, c+b\}$, $\{2+d, c+d\}$, $\{1+2, 1+c, a+2, a+c, b+2, b+c\}$, $\{1+1, 1+a, 1+b, a+1, a+a, a+b, b+1, b+a, b+b\}$, $\{1+d, a+d, b+d\}$, $\{d+2, d+c\}$, $\{d+1, d+a, d+b\}$, $\{d+d\}$

- **[13] : Transfer Point** $\Rightarrow d = 1$

  $\{2, c\}$, $\{1, a, b, d\}$, $\{2+2, 2+c, c+2, c+c\}$, $\{2+1, 2+a, 2+b, 2+d, c+1, c+a, c+b, c+d\}$, $\{1+2, 1+c, a+2, a+c, b+2, b+c, d+2, d+c\}$, $\{1+1, 1+a, 1+b, 1+d, a+1, a+a, a+b, a+d, b+1, b+a, b+b, b+d, d+1, d+a, d+b, d+d\}$

- **[14] : Confluence of** $\{5, 9, 13\}$

  $\{2\}$, $\{1, d\}$, $\{2+2\}$, $\{2+1, 2+d\}$, $\{1+2, d+2\}$, $\{1+1, 1+d, d+1, d+d\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{2+a\}$, $\{2+b\}$, $\{2+c\}$, $\{1+a, d+a\}$, $\{1+b, d+b\}$, $\{1+c, d+c\}$, $\{a+2\}$, $\{a+1, a+d\}$, $\{a+a\}$, $\{a+b\}$, $\{a+c\}$, $\{b+2\}$, $\{b+1, b+d\}$, $\{b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{c+2\}$, $\{c+1, c+d\}$, $\{c+a\}$, $\{c+b\}$, $\{c+c\}$

- **[15] : END**

  $\{2\}$, $\{1, d\}$, $\{2+2\}$, $\{2+1, 2+d\}$, $\{1+2, d+2\}$, $\{1+1, 1+d, d+1, d+d\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{2+a\}$, $\{2+b\}$, $\{2+c\}$, $\{1+a, d+a\}$, $\{1+b, d+b\}$, $\{1+c, d+c\}$, $\{a+2\}$, $\{a+1, a+d\}$, $\{a+a\}$, $\{a+b\}$, $\{a+c\}$, $\{b+2\}$, $\{b+1, b+d\}$, $\{b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{c+2\}$, $\{c+1, c+d\}$, $\{c+a\}$, $\{c+b\}$, $\{c+c\}$

5.7 Test Case 7

Here x and 0 must be equivalent at every program point after the assignment $x = 0$. Similarly, y and 0 must be equivalent after the assignment $y = 0$.

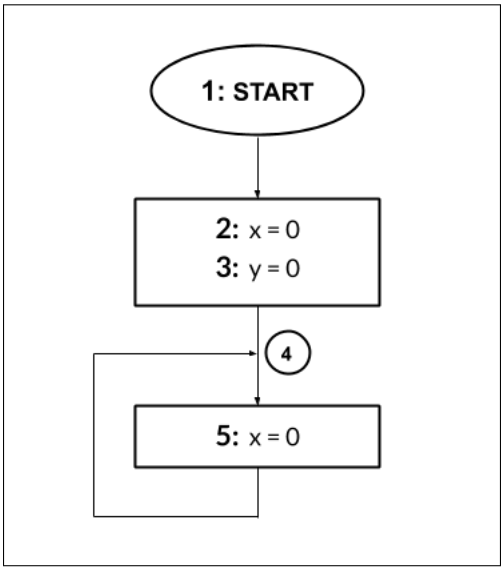

Figure 5.7 Test Case 7

- **[1] : START**

  $\{0\}$, $\{1\}$, $\{x\}$, $\{y\}$, $\{0+0\}$, $\{0+1\}$, $\{0+x\}$, $\{0+y\}$, $\{1+0\}$, $\{1+1\}$, $\{1+x\}$, $\{1+y\}$, $\{x+0\}$, $\{x+1\}$, $\{x+x\}$, $\{x+y\}$, $\{y+0\}$, $\{y+1\}$, $\{y+x\}$, $\{y+y\}$

- **[2] : Transfer Point** $\Rightarrow x = 0$

  $\{0, x\}$, $\{1\}$, $\{y\}$, $\{0+0, 0+x, x+0, x+x\}$, $\{0+1, x+1\}$, $\{0+y, x+y\}$, $\{1+0, 1+x\}$, $\{1+1\}$, $\{1+y\}$, $\{y+0, y+x\}$, $\{y+1\}$, $\{y+y\}$

- **[3] : Transfer Point** $\Rightarrow y = 0$

  $\{0, x, y\}$, $\{1\}$, $\{0+0, 0+x, 0+y, x+0, x+x, x+y, y+0, y+x, y+y\}$, $\{0+1, x+1, y+1\}$, $\{1+0, 1+x, 1+y\}$, $\{1+1\}$

- **[4] : Confluence of** $\{3, 5\}$

  $\{0, y\}$, $\{1\}$, $\{0+0, 0+y, y+0, y+y\}$, $\{0+1, y+1\}$, $\{1+0, 1+y\}$, $\{1+1\}$, $\{x\}$, $\{0+x, y+x\}$, $\{1+x\}$, $\{x+0, x+y\}$, $\{x+1\}$, $\{x+x\}$

- **[5] : Transfer Point** $\Rightarrow x = 1$

  $\{0, y\}$, $\{1, x\}$, $\{0+0, 0+y, y+0, y+y\}$, $\{0+1, 0+x, y+1, y+x\}$, $\{1+0, 1+y, x+0, x+y\}$, $\{1+1, 1+x, x+1, x+x\}$

5.8 Test Case 8

Here at the confluence point x, 0 and 1 must be mutually non-equivalent.

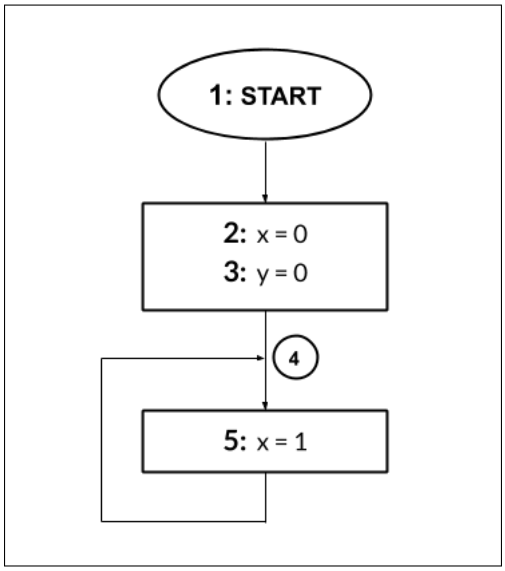

Figure 5.8 Test Case 8

- **[1] : START**

  $\{0\}$, $\{1\}$, $\{x\}$, $\{y\}$, $\{0+0\}$, $\{0+1\}$, $\{0+x\}$, $\{0+y\}$, $\{1+0\}$, $\{1+1\}$, $\{1+x\}$, $\{1+y\}$, $\{x+0\}$, $\{x+1\}$, $\{x+x\}$, $\{x+y\}$, $\{y+0\}$, $\{y+1\}$, $\{y+x\}$, $\{y+y\}$

- **[2] : Transfer Point** $\Rightarrow x = 0$

  $\{0,x\}$, $\{1\}$, $\{y\}$, $\{0+0, 0+x, x+0, x+x\}$, $\{0+1, x+1\}$, $\{0+y, x+y\}$, $\{1+0, 1+x\}$, $\{1+1\}$, $\{1+y\}$, $\{y+0, y+x\}$, $\{y+1\}$, $\{y+y\}$

- **[3] : Transfer Point** $\Rightarrow y = 0$

  $\{0,x,y\}$, $\{1\}$, $\{0+0, 0+x, 0+y, x+0, x+x, x+y, y+0, y+x, y+y\}$, $\{0+1, x+1, y+1\}$, $\{1+0, 1+x, 1+y\}$, $\{1+1\}$

- **[4] : Confluence of** $\{3,5\}$

  $\{0,y\}$, $\{1\}$, $\{0+0, 0+y, y+0, y+y\}$, $\{0+1, y+1\}$, $\{1+0, 1+y\}$, $\{1+1\}$, $\{x\}$, $\{0+x, y+x\}$, $\{1+x\}$, $\{x+0, x+y\}$, $\{x+1\}$, $\{x+x\}$

- **[5] : Transfer Point** $\Rightarrow x = 1$

  $\{0, y\}$, $\{1, x\}$, $\{0{+}0, 0{+}y, y{+}0, y{+}y\}$, $\{0{+}1, 0{+}x, y{+}1, y{+}x\}$, $\{1{+}0, 1{+}y, x{+}0, x{+}y\}$, $\{1 + 1, 1 + x, x + 1, x + x\}$

5.9 Test Case 9

Here at the program point after assignment $z = x + 1$, y and z must be equivalent.

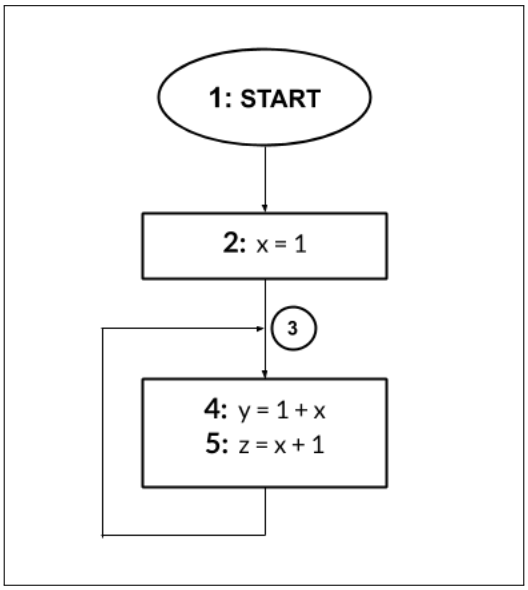

Figure 5.9 Test Case 9

- **[1] : START**

  $\{1\}$, $\{x\}$, $\{y\}$, $\{z\}$, $\{1 + 1\}$, $\{1 + x\}$, $\{1 + y\}$, $\{1 + z\}$, $\{x + 1\}$, $\{x + x\}$, $\{x + y\}$, $\{x + z\}$, $\{y + 1\}$, $\{y + x\}$, $\{y + y\}$, $\{y + z\}$, $\{z + 1\}$, $\{z + x\}$, $\{z + y\}$, $\{z + z\}$

- **[2] : Transfer Point** $\Rightarrow x = 1$

  $\{1, x\}$, $\{y\}$, $\{z\}$, $\{1{+}1, 1{+}x, x{+}1, x{+}x\}$, $\{1{+}y, x{+}y\}$, $\{1{+}z, x{+}z\}$, $\{y{+}1, y{+}x\}$, $\{y + y\}$, $\{y + z\}$, $\{z + 1, z + x\}$, $\{z + y\}$, $\{z + z\}$

- **[3] : Confluence of** $\{2, 5\}$

  $\{1, x\}$, $\{1{+}1, 1{+}x, x{+}1, x{+}x\}$, $\{y\}$, $\{z\}$, $\{1{+}y, x{+}y\}$, $\{1{+}z, x{+}z\}$, $\{y{+}1, y{+}x\}$, $\{y + y\}$, $\{y + z\}$, $\{z + 1, z + x\}$, $\{z + y\}$, $\{z + z\}$

- **[4] : Transfer Point** $\Rightarrow y = 1 + x$

  $\{1, x\}$, $\{y, 1+1, 1+x, x+1, x+x\}$, $\{1+y, x+y\}$, $\{y+1, y+x\}$, $\{y+y\}$, $\{z\}$, $\{1+z, x+z\}$, $\{z+1, z+x\}$, $\{z+z\}$, $\{y+z\}$, $\{z+y\}$

- **[5] : Transfer Point** $\Rightarrow z = x + 1$

  $\{1, x\}$, $\{y, z, 1+1, 1+x, x+1, x+x\}$, $\{1+y, 1+z, x+y, x+z\}$, $\{y+1, y+x, z+1, z+x\}$, $\{y+y, y+z, z+y, z+z\}$

5.10 Test Case 10

A case that involves two intersecting loops. Such a control flow graph can result from conditional jumps to two different labels. Example - `if(x < y) goto L1 else goto L2`.

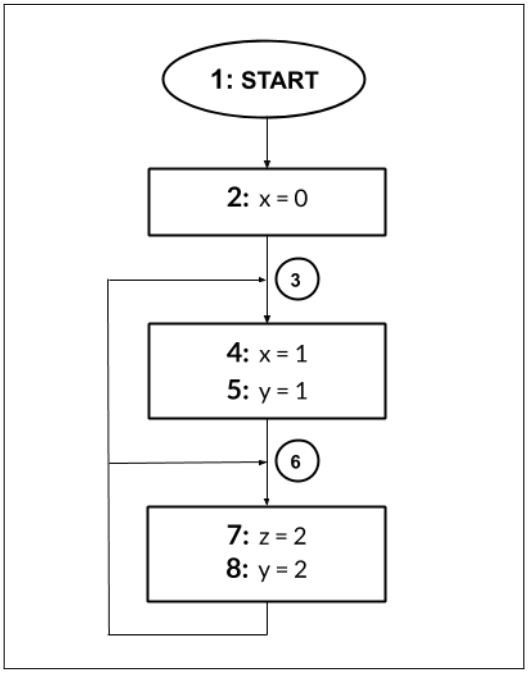

Figure 5.10 Test Case 10

- **[1] : START**

  $\{0\}$, $\{1\}$, $\{2\}$, $\{x\}$, $\{y\}$, $\{z\}$, $\{0+0\}$, $\{0+1\}$, $\{0+2\}$, $\{0+x\}$, $\{0+y\}$, $\{0+z\}$, $\{1+0\}$, $\{1+1\}$, $\{1+2\}$, $\{1+x\}$, $\{1+y\}$, $\{1+z\}$, $\{2+0\}$, $\{2+1\}$, $\{2+2\}$, $\{2+x\}$,

$\{2+y\}$, $\{2+z\}$, $\{x+0\}$, $\{x+1\}$, $\{x+2\}$, $\{x+x\}$, $\{x+y\}$, $\{x+z\}$, $\{y+0\}$, $\{y+1\}$, $\{y+2\}$, $\{y+x\}$, $\{y+y\}$, $\{y+z\}$, $\{z+0\}$, $\{z+1\}$, $\{z+2\}$, $\{z+x\}$, $\{z+y\}$, $\{z+z\}$

- **[2] : Transfer Point** $\Rightarrow x = 0$

  $\{0, x\}$, $\{1\}$, $\{2\}$, $\{y\}$, $\{z\}$, $\{0+0, 0+x, x+0, x+x\}$, $\{0+1, x+1\}$, $\{0+2, x+2\}$, $\{0+y, x+y\}$, $\{0+z, x+z\}$, $\{1+0, 1+x\}$, $\{1+1\}$, $\{1+2\}$, $\{1+y\}$, $\{1+z\}$, $\{2+0, 2+x\}$, $\{2+1\}$, $\{2+2\}$, $\{2+y\}$, $\{2+z\}$, $\{y+0, y+x\}$, $\{y+1\}$, $\{y+2\}$, $\{y+y\}$, $\{y+z\}$, $\{z+0, z+x\}$, $\{z+1\}$, $\{z+2\}$, $\{z+y\}$, $\{z+z\}$

- **[3] : Confluence of** $\{2, 8\}$

  $\{0\}$, $\{1\}$, $\{2\}$, $\{0+0\}$, $\{0+1\}$, $\{0+2\}$, $\{1+0\}$, $\{1+1\}$, $\{1+2\}$, $\{2+0\}$, $\{2+1\}$, $\{2+2\}$, $\{x\}$, $\{y\}$, $\{z\}$, $\{0+x\}$, $\{0+y\}$, $\{0+z\}$, $\{1+x\}$, $\{1+y\}$, $\{1+z\}$, $\{2+x\}$, $\{2+y\}$, $\{2+z\}$, $\{x+0\}$, $\{x+1\}$, $\{x+2\}$, $\{x+x\}$, $\{x+y\}$, $\{x+z\}$, $\{y+0\}$, $\{y+1\}$, $\{y+2\}$, $\{y+x\}$, $\{y+y\}$, $\{y+z\}$, $\{z+0\}$, $\{z+1\}$, $\{z+2\}$, $\{z+x\}$, $\{z+y\}$, $\{z+z\}$

- **[4] : Transfer Point** $\Rightarrow x = 1$

  $\{0\}$, $\{1, x\}$, $\{2\}$, $\{0+0\}$, $\{0+1, 0+x\}$, $\{0+2\}$, $\{1+0, x+0\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+2, x+2\}$, $\{2+0\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{y\}$, $\{z\}$, $\{0+y\}$, $\{0+z\}$, $\{1+y, x+y\}$, $\{1+z, x+z\}$, $\{2+y\}$, $\{2+z\}$, $\{y+0\}$, $\{y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$, $\{y+z\}$, $\{z+0\}$, $\{z+1, z+x\}$, $\{z+2\}$, $\{z+y\}$, $\{z+z\}$

- **[5] : Transfer Point** $\Rightarrow y = 1$

  $\{0\}$, $\{1, x, y\}$, $\{2\}$, $\{0+0\}$, $\{0+1, 0+x, 0+y\}$, $\{0+2\}$, $\{1+0, x+0, y+0\}$, $\{1+1, 1+x, 1+y, x+1, x+x, x+y, y+1, y+x, y+y\}$, $\{1+2, x+2, y+2\}$, $\{2+0\}$, $\{2+1, 2+x, 2+y\}$, $\{2+2\}$, $\{z\}$, $\{0+z\}$, $\{1+z, x+z, y+z\}$, $\{2+z\}$, $\{z+0\}$, $\{z+1, z+x, z+y\}$, $\{z+2\}$, $\{z+z\}$

- **[6] : Confluence of** $\{5, 8\}$

  $\{0\}$, $\{1, x\}$, $\{2\}$, $\{0+0\}$, $\{0+1, 0+x\}$, $\{0+2\}$, $\{1+0, x+0\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+2, x+2\}$, $\{2+0\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{y\}$, $\{z\}$, $\{0+y\}$, $\{0+z\}$, $\{1+y, x+y\}$,

$\{1+z, x+z\}$, $\{2+y\}$, $\{2+z\}$, $\{y+0\}$, $\{y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$, $\{y+z\}$, $\{z+0\}$, $\{z+1, z+x\}$, $\{z+2\}$, $\{z+y\}$, $\{z+z\}$

- **[7] : Transfer Point** $\Rightarrow z = 2$

  $\{0\}$, $\{1, x\}$, $\{2, z\}$, $\{0+0\}$, $\{0+1, 0+x\}$, $\{0+2, 0+z\}$, $\{1+0, x+0\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+2, 1+z, x+2, x+z\}$, $\{2+0, z+0\}$, $\{2+1, 2+x, z+1, z+x\}$, $\{2+2, 2+z, z+2, z+z\}$, $\{y\}$, $\{0+y\}$, $\{1+y, x+y\}$, $\{2+y, z+y\}$, $\{y+0\}$, $\{y+1, y+x\}$, $\{y+2, y+z\}$, $\{y+y\}$

- **[8] : Transfer Point** $\Rightarrow y = 2$

  $\{0\}$, $\{1, x\}$, $\{2, y, z\}$, $\{0+0\}$, $\{0+1, 0+x\}$, $\{0+2, 0+y, 0+z\}$, $\{1+0, x+0\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+2, 1+y, 1+z, x+2, x+y, x+z\}$, $\{2+0, y+0, z+0\}$, $\{2+1, 2+x, y+1, y+x, z+1, z+x\}$, $\{2+2, 2+y, 2+z, y+2, y+y, y+z, z+2, z+y, z+z\}$

5.11 Test Case 11

Here at the **END** program point, x and y must be equivalent but they must not be equivalent to either of 1 and 2.

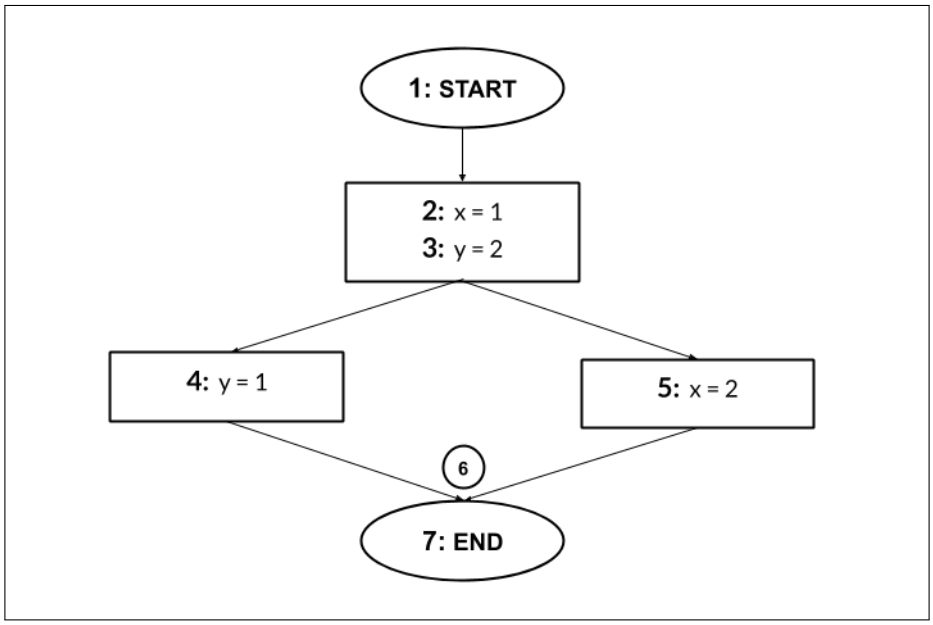

Figure 5.11 Test Case 11

- **[1] : START**

  $\{1\}$, $\{2\}$, $\{x\}$, $\{y\}$, $\{1+1\}$, $\{1+2\}$, $\{1+x\}$, $\{1+y\}$, $\{2+1\}$, $\{2+2\}$, $\{2+x\}$, $\{2+y\}$, $\{x+1\}$, $\{x+2\}$, $\{x+x\}$, $\{x+y\}$, $\{y+1\}$, $\{y+2\}$, $\{y+x\}$, $\{y+y\}$

- **[2] : Transfer Point** $\Rightarrow x = 1$

  $\{1, x\}$, $\{2\}$, $\{y\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+2, x+2\}$, $\{1+y, x+y\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{2+y\}$, $\{y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$

- **[3] : Transfer Point** $\Rightarrow y = 2$

  $\{1, x\}$, $\{2, y\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+2, 1+y, x+2, x+y\}$, $\{2+1, 2+x, y+1, y+x\}$, $\{2+2, 2+y, y+2, y+y\}$

- **[4] : Transfer Point** $\Rightarrow y = 1$

  $\{1, x, y\}$, $\{2\}$, $\{1+1, 1+x, 1+y, x+1, x+x, x+y, y+1, y+x, y+y\}$, $\{1+2, x+2, y+2\}$, $\{2+1, 2+x, 2+y\}$, $\{2+2\}$

- **[5] : Transfer Point** $\Rightarrow x = 2$

  $\{1\}$, $\{2, x, y\}$, $\{1+1\}$, $\{1+2, 1+x, 1+y\}$, $\{2+1, x+1, y+1\}$, $\{2+2, 2+x, 2+y, x+2, x+x, x+y, y+2, y+x, y+y\}$

- **[6] : Confluence of** $\{4, 5\}$

  $\{1\}$, $\{2\}$, $\{1+1\}$, $\{1+2\}$, $\{2+1\}$, $\{2+2\}$, $\{x, y\}$, $\{1+x, 1+y\}$, $\{2+x, 2+y\}$, $\{x+1, y+1\}$, $\{x+2, y+2\}$, $\{x+x, x+y, y+x, y+y\}$

- **[7] : END**

  $\{1\}$, $\{2\}$, $\{1+1\}$, $\{1+2\}$, $\{2+1\}$, $\{2+2\}$, $\{x, y\}$, $\{1+x, 1+y\}$, $\{2+x, 2+y\}$, $\{x+1, y+1\}$, $\{x+2, y+2\}$, $\{x+x, x+y, y+x, y+y\}$

5.12 Test Case 12

Here even though the expressions x and y changes in every iteration, the algorithm must terminate in finite steps.

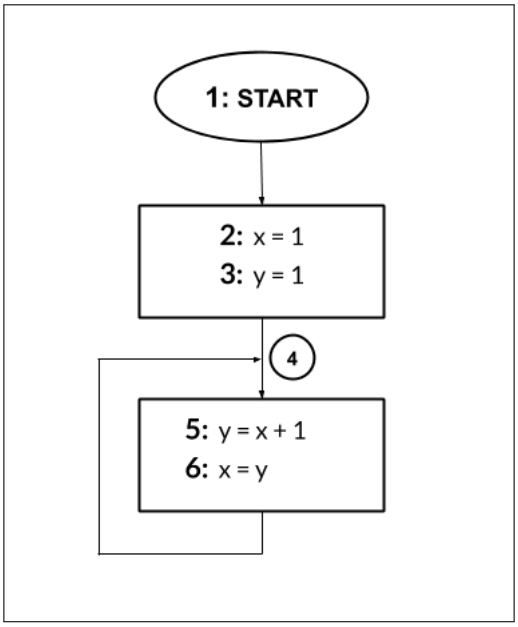

Figure 5.12 Test Case 12

- **[1] : START**

  $\{1\}$, $\{x\}$, $\{y\}$, $\{1+1\}$, $\{1+x\}$, $\{1+y\}$, $\{x+1\}$, $\{x+x\}$, $\{x+y\}$, $\{y+1\}$, $\{y+x\}$, $\{y+y\}$

- **[2] : Transfer Point** $\Rightarrow x = 1$

  $\{1, x\}$, $\{y\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+y, x+y\}$, $\{y+1, y+x\}$, $\{y+y\}$

- **[3] : Transfer Point** $\Rightarrow y = 1$

  $\{1, x, y\}$, $\{1+1, 1+x, 1+y, x+1, x+x, x+y, y+1, y+x, y+y\}$

- **[4] : Confluence of** $\{3, 6\}$

  $\{1\}$, $\{1+1\}$, $\{x, y\}$, $\{1+x, 1+y\}$, $\{x+1, y+1\}$, $\{x+x, x+y, y+x, y+y\}$

- **[5] : Transfer Point** $\Rightarrow y = x + 1$

  $\{1\}$, $\{1+1\}$, $\{x\}$, $\{1+x\}$, $\{y, x+1\}$, $\{x+x\}$, $\{1+y\}$, $\{x+y\}$, $\{y+1\}$, $\{y+x\}$, $\{y+y\}$

- **[6] : Transfer Point** $\Rightarrow x = y$

  $\{1\}$, $\{1+1\}$, $\{x, y\}$, $\{1+x, 1+y\}$, $\{x+1, y+1\}$, $\{x+x, x+y, y+x, y+y\}$

5.13 Test Case 13

Here the value assigned to x on both the branches are same, and the algorithm must acknowledge this fact if it takes the assigned values into account.

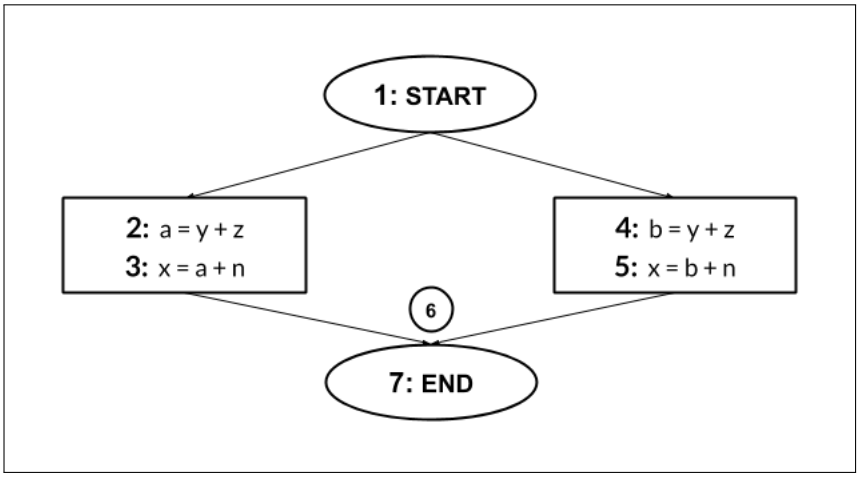

Figure 5.13 Test Case 13

- **[1] : START**

  $\{a\}$, $\{y\}$, $\{z\}$, $\{x\}$, $\{n\}$, $\{b\}$, $\{a+a\}$, $\{a+y\}$, $\{a+z\}$, $\{a+x\}$, $\{a+n\}$, $\{a+b\}$, $\{y+a\}$, $\{y+y\}$, $\{y+z\}$, $\{y+x\}$, $\{y+n\}$, $\{y+b\}$, $\{z+a\}$, $\{z+y\}$, $\{z+z\}$, $\{z+x\}$, $\{z+n\}$, $\{z+b\}$, $\{x+a\}$, $\{x+y\}$, $\{x+z\}$, $\{x+x\}$, $\{x+n\}$, $\{x+b\}$, $\{n+a\}$, $\{n+y\}$, $\{n+z\}$, $\{n+x\}$, $\{n+n\}$, $\{n+b\}$, $\{b+a\}$, $\{b+y\}$, $\{b+z\}$, $\{b+x\}$, $\{b+n\}$, $\{b+b\}$

- **[2] : Transfer Point** $\Rightarrow a = y + z$

  $\{y\}$, $\{z\}$, $\{x\}$, $\{n\}$, $\{b\}$, $\{y+y\}$, $\{a, y+z\}$, $\{y+x\}$, $\{y+n\}$, $\{y+b\}$, $\{z+y\}$, $\{z+z\}$, $\{z+x\}$, $\{z+n\}$, $\{z+b\}$, $\{x+y\}$, $\{x+z\}$, $\{x+x\}$, $\{x+n\}$, $\{x+b\}$, $\{n+y\}$, $\{n+z\}$, $\{n+x\}$, $\{n+n\}$, $\{n+b\}$, $\{b+y\}$, $\{b+z\}$, $\{b+x\}$, $\{b+n\}$, $\{b+b\}$, $\{a+a\}$, $\{a+y\}$, $\{a+z\}$, $\{a+x\}$, $\{a+n\}$, $\{a+b\}$, $\{y+a\}$, $\{z+a\}$, $\{x+a\}$, $\{n+a\}$, $\{b+a\}$

- **[3] : Transfer Point** $\Rightarrow x = a + n$

  $\{y\}$, $\{z\}$, $\{n\}$, $\{b\}$, $\{y+y\}$, $\{a, y+z\}$, $\{y+n\}$, $\{y+b\}$, $\{z+y\}$, $\{z+z\}$, $\{z+n\}$, $\{z+b\}$, $\{n+y\}$, $\{n+z\}$, $\{n+n\}$, $\{n+b\}$, $\{b+y\}$, $\{b+z\}$, $\{b+n\}$, $\{b+b\}$, $\{a+a\}$,

$\{a+y\}$, $\{a+z\}$, $\{x, a+n\}$, $\{a+b\}$, $\{y+a\}$, $\{z+a\}$, $\{n+a\}$, $\{b+a\}$, $\{a+x\}$,
$\{y+x\}$, $\{z+x\}$, $\{x+a\}$, $\{x+y\}$, $\{x+z\}$, $\{x+x\}$, $\{x+n\}$, $\{x+b\}$, $\{n+x\}$, $\{b+x\}$

- **[4] : Transfer Point** $\Rightarrow b = y + z$

  $\{a\}$, $\{y\}$, $\{z\}$, $\{x\}$, $\{n\}$, $\{a+a\}$, $\{a+y\}$, $\{a+z\}$, $\{a+x\}$, $\{a+n\}$, $\{y+a\}$, $\{y+y\}$,
  $\{b, y+z\}$, $\{y+x\}$, $\{y+n\}$, $\{z+a\}$, $\{z+y\}$, $\{z+z\}$, $\{z+x\}$, $\{z+n\}$, $\{x+a\}$,
  $\{x+y\}$, $\{x+z\}$, $\{x+x\}$, $\{x+n\}$, $\{n+a\}$, $\{n+y\}$, $\{n+z\}$, $\{n+x\}$, $\{n+n\}$,
  $\{b+b\}$, $\{b+y\}$, $\{b+z\}$, $\{b+x\}$, $\{b+n\}$, $\{y+b\}$, $\{z+b\}$, $\{x+b\}$, $\{n+b\}$, $\{a+b\}$,
  $\{b+a\}$

- **[5] : Transfer Point** $\Rightarrow x = b + n$

  $\{a\}$, $\{y\}$, $\{z\}$, $\{n\}$, $\{a+a\}$, $\{a+y\}$, $\{a+z\}$, $\{a+n\}$, $\{y+a\}$, $\{y+y\}$, $\{b, y+z\}$,
  $\{y+n\}$, $\{z+a\}$, $\{z+y\}$, $\{z+z\}$, $\{z+n\}$, $\{n+a\}$, $\{n+y\}$, $\{n+z\}$, $\{n+n\}$, $\{b+b\}$,
  $\{b+y\}$, $\{b+z\}$, $\{x, b+n\}$, $\{y+b\}$, $\{z+b\}$, $\{n+b\}$, $\{b+x\}$, $\{y+x\}$, $\{z+x\}$,
  $\{x+b\}$, $\{x+y\}$, $\{x+z\}$, $\{x+x\}$, $\{x+n\}$, $\{n+x\}$, $\{a+b\}$, $\{b+a\}$, $\{a+x\}$, $\{x+a\}$

- **[6] : Confluence of** $\{3, 5\}$

  $\{y\}$, $\{z\}$, $\{n\}$, $\{y+y\}$, $\{y+z\}$, $\{y+n\}$, $\{z+y\}$, $\{z+z\}$, $\{z+n\}$, $\{n+y\}$, $\{n+z\}$,
  $\{n+n\}$, $\{x\}$, $\{y+x\}$, $\{z+x\}$, $\{x+y\}$, $\{x+z\}$, $\{x+x\}$, $\{x+n\}$, $\{n+x\}$, $\{a\}$, $\{b\}$,
  $\{a+a\}$, $\{a+y\}$, $\{a+z\}$, $\{a+x\}$, $\{a+n\}$, $\{a+b\}$, $\{y+a\}$, $\{y+b\}$, $\{z+a\}$, $\{z+b\}$,
  $\{x+a\}$, $\{x+b\}$, $\{n+a\}$, $\{n+b\}$, $\{b+a\}$, $\{b+y\}$, $\{b+z\}$, $\{b+x\}$, $\{b+n\}$, $\{b+b\}$

- **[7] : END**

  $\{y\}$, $\{z\}$, $\{n\}$, $\{y+y\}$, $\{y+z\}$, $\{y+n\}$, $\{z+y\}$, $\{z+z\}$, $\{z+n\}$, $\{n+y\}$, $\{n+z\}$,
  $\{n+n\}$, $\{x\}$, $\{y+x\}$, $\{z+x\}$, $\{x+y\}$, $\{x+z\}$, $\{x+x\}$, $\{x+n\}$, $\{n+x\}$, $\{a\}$, $\{b\}$,
  $\{a+a\}$, $\{a+y\}$, $\{a+z\}$, $\{a+x\}$, $\{a+n\}$, $\{a+b\}$, $\{y+a\}$, $\{y+b\}$, $\{z+a\}$, $\{z+b\}$,
  $\{x+a\}$, $\{x+b\}$, $\{n+a\}$, $\{n+b\}$, $\{b+a\}$, $\{b+y\}$, $\{b+z\}$, $\{b+x\}$, $\{b+n\}$, $\{b+b\}$

5.14 Test Case 14

Here at the **END** program point x and x_3 must be equivalent.

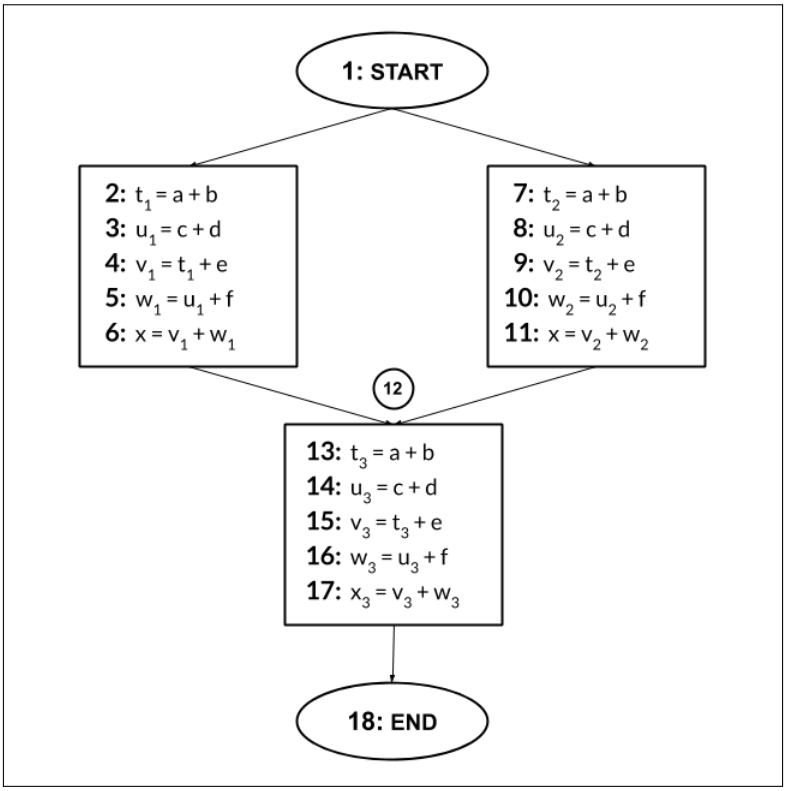

Figure 5.14 Test Case 14

The complete equivalence information for this test case is very long and hence only the partition at `END` point is given. Refer to GitHub for complete results.

- **[18] : END**

  $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{e\}$, $\{f\}$, $\{a+a\}$, $\{t_3, a+b\}$, $\{a+c\}$, $\{a+d\}$, $\{a+e\}$, $\{a+f\}$, $\{b+a\}$, $\{b+b\}$, $\{b+c\}$, $\{b+d\}$, $\{b+e\}$, $\{b+f\}$, $\{c+a\}$, $\{c+b\}$, $\{c+c\}$, $\{u_3, c+d\}$, $\{c+e\}$, $\{c+f\}$, $\{d+a\}$, $\{d+b\}$, $\{d+c\}$, $\{d+d\}$, $\{d+e\}$, $\{d+f\}$, $\{e+a\}$, $\{e+b\}$, $\{e+c\}$, $\{e+d\}$, $\{e+e\}$, $\{e+f\}$, $\{f+a\}$, $\{f+b\}$, $\{f+c\}$, $\{f+d\}$, $\{f+e\}$, $\{f+f\}$, $\{t_3+t_3\}$, $\{t_3+a\}$, $\{t_3+b\}$, $\{t_3+c\}$, $\{t_3+d\}$, $\{v_3, t_3+e\}$, $\{t_3+f\}$, $\{a+t_3\}$, $\{b+t_3\}$, $\{c+t_3\}$, $\{d+t_3\}$, $\{e+t_3\}$, $\{f+t_3\}$, $\{t_3+u_3\}$, $\{a+u_3\}$, $\{b+u_3\}$, $\{u_3+t_3\}$, $\{u_3+a\}$, $\{u_3+b\}$, $\{u_3+u_3\}$, $\{u_3+c\}$, $\{u_3+d\}$, $\{u_3+e\}$, $\{w_3, u_3+f\}$, $\{c+u_3\}$, $\{d+u_3\}$,

$\{e+u_3\}$, $\{f+u_3\}$, $\{t_3+v_3\}$, $\{a+v_3\}$, $\{b+v_3\}$, $\{u_3+v_3\}$, $\{c+v_3\}$, $\{d+v_3\}$, $\{v_3+t_3\}$, $\{v_3+a\}$, $\{v_3+b\}$, $\{v_3+u_3\}$, $\{v_3+c\}$, $\{v_3+d\}$, $\{v_3+v_3\}$, $\{v_3+e\}$, $\{v_3+f\}$, $\{e+v_3\}$, $\{f+v_3\}$, $\{t_3+w_3\}$, $\{a+w_3\}$, $\{b+w_3\}$, $\{u_3+w_3\}$, $\{c+w_3\}$, $\{d+w_3\}$, $\{x, x_3, v_3+w_3\}$, $\{e+w_3\}$, $\{w_3+t_3\}$, $\{w_3+a\}$, $\{w_3+b\}$, $\{w_3+u_3\}$, $\{w_3+c\}$, $\{w_3+d\}$, $\{w_3+v_3\}$, $\{w_3+e\}$, $\{w_3+w_3\}$, $\{w_3+f\}$, $\{f+w_3\}$, $\{t_3+x, t_3+x_3\}$, $\{a+x, a+x_3\}$, $\{b+x, b+x_3\}$, $\{u_3+x, u_3+x_3\}$, $\{c+x, c+x_3\}$, $\{d+x, d+x_3\}$, $\{v_3+x, v_3+x_3\}$, $\{e+x, e+x_3\}$, $\{w_3+x, w_3+x_3\}$, $\{f+x, f+x_3\}$, $\{x+t_3, x_3+t_3\}$, $\{x+a, x_3+a\}$, $\{x+b, x_3+b\}$, $\{x+u_3, x_3+u_3\}$, $\{x+c, x_3+c\}$, $\{x+d, x_3+d\}$, $\{x+v_3, x_3+v_3\}$, $\{x+e, x_3+e\}$, $\{x+w_3, x_3+w_3\}$, $\{x+f, x_3+f\}$, $\{x+x, x+x_3, x_3+x, x_3+x_3\}$, $\{t_1\}$, $\{u_1\}$, $\{v_1\}$, $\{w_1\}$, $\{t_2\}$, $\{u_2\}$, $\{v_2\}$, $\{w_2\}$, $\{t_1+t_1\}$, $\{t_1+a\}$, $\{t_1+b\}$, $\{t_1+u_1\}$, $\{t_1+c\}$, $\{t_1+d\}$, $\{t_1+v_1\}$, $\{t_1+e\}$, $\{t_1+w_1\}$, $\{t_1+f\}$, $\{t_1+x, t_1+x_3\}$, $\{t_1+t_2\}$, $\{t_1+u_2\}$, $\{t_1+v_2\}$, $\{t_1+w_2\}$, $\{a+t_1\}$, $\{a+u_1\}$, $\{a+v_1\}$, $\{a+w_1\}$, $\{a+t_2\}$, $\{a+u_2\}$, $\{a+v_2\}$, $\{a+w_2\}$, $\{b+t_1\}$, $\{b+u_1\}$, $\{b+v_1\}$, $\{b+w_1\}$, $\{b+t_2\}$, $\{b+u_2\}$, $\{b+v_2\}$, $\{b+w_2\}$, $\{u_1+t_1\}$, $\{u_1+a\}$, $\{u_1+b\}$, $\{u_1+u_1\}$, $\{u_1+c\}$, $\{u_1+d\}$, $\{u_1+v_1\}$, $\{u_1+e\}$, $\{u_1+w_1\}$, $\{u_1+f\}$, $\{u_1+x, u_1+x_3\}$, $\{u_1+t_2\}$, $\{u_1+u_2\}$, $\{u_1+v_2\}$, $\{u_1+w_2\}$, $\{c+t_1\}$, $\{c+u_1\}$, $\{c+v_1\}$, $\{c+w_1\}$, $\{c+t_2\}$, $\{c+u_2\}$, $\{c+v_2\}$, $\{c+w_2\}$, $\{d+t_1\}$, $\{d+u_1\}$, $\{d+v_1\}$, $\{d+w_1\}$, $\{d+t_2\}$, $\{d+u_2\}$, $\{d+v_2\}$, $\{d+w_2\}$, $\{v_1+t_1\}$, $\{v_1+a\}$, $\{v_1+b\}$, $\{v_1+u_1\}$, $\{v_1+c\}$, $\{v_1+d\}$, $\{v_1+v_1\}$, $\{v_1+e\}$, $\{v_1+w_1\}$, $\{v_1+f\}$, $\{v_1+x, v_1+x_3\}$, $\{v_1+t_2\}$, $\{v_1+u_2\}$, $\{v_1+v_2\}$, $\{v_1+w_2\}$, $\{e+t_1\}$, $\{e+u_1\}$, $\{e+v_1\}$, $\{e+w_1\}$, $\{e+t_2\}$, $\{e+u_2\}$, $\{e+v_2\}$, $\{e+w_2\}$, $\{w_1+t_1\}$, $\{w_1+a\}$, $\{w_1+b\}$, $\{w_1+u_1\}$, $\{w_1+c\}$, $\{w_1+d\}$, $\{w_1+v_1\}$, $\{w_1+e\}$, $\{w_1+w_1\}$, $\{w_1+f\}$, $\{w_1+x, w_1+x_3\}$, $\{w_1+t_2\}$, $\{w_1+u_2\}$, $\{w_1+v_2\}$, $\{w_1+w_2\}$, $\{f+t_1\}$, $\{f+u_1\}$, $\{f+v_1\}$, $\{f+w_1\}$, $\{f+t_2\}$, $\{f+u_2\}$, $\{f+v_2\}$, $\{f+w_2\}$, $\{x+t_1, x_3+t_1\}$, $\{x+u_1, x_3+u_1\}$, $\{x+v_1, x_3+v_1\}$, $\{x+w_1, x_3+w_1\}$, $\{x+t_2, x_3+t_2\}$, $\{x+u_2, x_3+u_2\}$, $\{x+v_2, x_3+v_2\}$, $\{x+w_2, x_3+w_2\}$, $\{t_2+t_1\}$, $\{t_2+a\}$, $\{t_2+b\}$, $\{t_2+u_1\}$, $\{t_2+c\}$, $\{t_2+d\}$, $\{t_2+v_1\}$, $\{t_2+e\}$, $\{t_2+w_1\}$, $\{t_2+f\}$, $\{t_2+x, t_2+x_3\}$, $\{t_2+t_2\}$, $\{t_2+u_2\}$, $\{t_2+v_2\}$, $\{t_2+w_2\}$, $\{u_2+t_1\}$, $\{u_2+a\}$, $\{u_2+b\}$, $\{u_2+u_1\}$, $\{u_2+c\}$, $\{u_2+d\}$, $\{u_2+v_1\}$, $\{u_2+e\}$, $\{u_2+w_1\}$, $\{u_2+f\}$,

$\{u_2 + x, u_2 + x_3\}$, $\{u_2 + t_2\}$, $\{u_2 + u_2\}$, $\{u_2 + v_2\}$, $\{u_2 + w_2\}$, $\{v_2 + t_1\}$, $\{v_2 + a\}$, $\{v_2 + b\}$, $\{v_2 + u_1\}$, $\{v_2 + c\}$, $\{v_2 + d\}$, $\{v_2 + v_1\}$, $\{v_2 + e\}$, $\{v_2 + w_1\}$, $\{v_2 + f\}$, $\{v_2 + x, v_2 + x_3\}$, $\{v_2 + t_2\}$, $\{v_2 + u_2\}$, $\{v_2 + v_2\}$, $\{v_2 + w_2\}$, $\{w_2 + t_1\}$, $\{w_2 + a\}$, $\{w_2 + b\}$, $\{w_2 + u_1\}$, $\{w_2 + c\}$, $\{w_2 + d\}$, $\{w_2 + v_1\}$, $\{w_2 + e\}$, $\{w_2 + w_1\}$, $\{w_2 + f\}$, $\{w_2 + x, w_2 + x_3\}$, $\{w_2 + t_2\}$, $\{w_2 + u_2\}$, $\{w_2 + v_2\}$, $\{w_2 + w_2\}$, $\{t_1 + t_3\}$, $\{u_1 + t_3\}$, $\{v_1 + t_3\}$, $\{w_1 + t_3\}$, $\{t_2 + t_3\}$, $\{u_2 + t_3\}$, $\{v_2 + t_3\}$, $\{w_2 + t_3\}$, $\{t_3 + t_1\}$, $\{t_3 + u_1\}$, $\{t_3 + v_1\}$, $\{t_3 + w_1\}$, $\{t_3 + t_2\}$, $\{t_3 + u_2\}$, $\{t_3 + v_2\}$, $\{t_3 + w_2\}$, $\{t_1 + u_3\}$, $\{u_1 + u_3\}$, $\{v_1 + u_3\}$, $\{w_1 + u_3\}$, $\{t_2 + u_3\}$, $\{u_2 + u_3\}$, $\{v_2 + u_3\}$, $\{w_2 + u_3\}$, $\{u_3 + t_1\}$, $\{u_3 + u_1\}$, $\{u_3 + v_1\}$, $\{u_3 + w_1\}$, $\{u_3 + t_2\}$, $\{u_3 + u_2\}$, $\{u_3 + v_2\}$, $\{u_3 + w_2\}$, $\{t_1 + v_3\}$, $\{u_1 + v_3\}$, $\{v_1 + v_3\}$, $\{w_1 + v_3\}$, $\{t_2 + v_3\}$, $\{u_2 + v_3\}$, $\{v_2 + v_3\}$, $\{w_2 + v_3\}$, $\{v_3 + t_1\}$, $\{v_3 + u_1\}$, $\{v_3 + v_1\}$, $\{v_3 + w_1\}$, $\{v_3 + t_2\}$, $\{v_3 + u_2\}$, $\{v_3 + v_2\}$, $\{v_3 + w_2\}$, $\{t_1 + w_3\}$, $\{u_1 + w_3\}$, $\{v_1 + w_3\}$, $\{w_1 + w_3\}$, $\{t_2 + w_3\}$, $\{u_2 + w_3\}$, $\{v_2 + w_3\}$, $\{w_2 + w_3\}$, $\{w_3 + t_1\}$, $\{w_3 + u_1\}$, $\{w_3 + v_1\}$, $\{w_3 + w_1\}$, $\{w_3 + t_2\}$, $\{w_3 + u_2\}$, $\{w_3 + v_2\}$, $\{w_3 + w_2\}$

5.15 Test Case 15

Here at the **END** program point y and $x + 2$ must be equivalent.

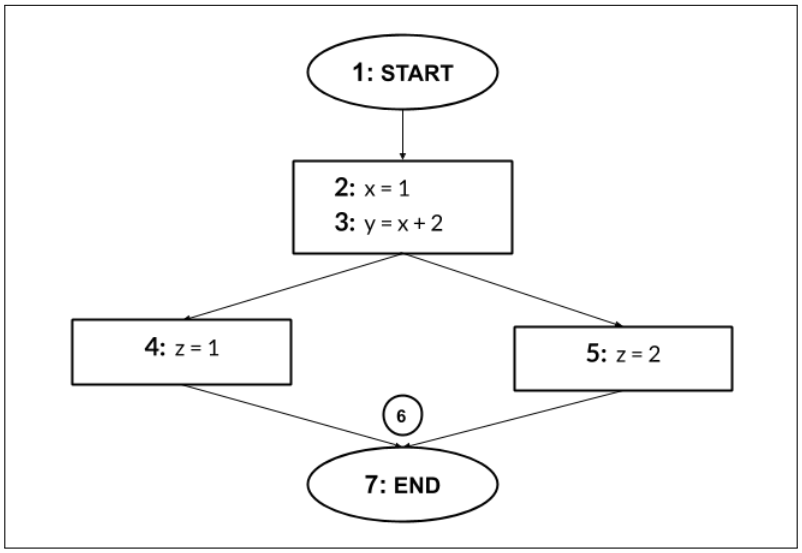

Figure 5.15 Test Case 15

- **[1] : START**

  $\{1\}$, $\{2\}$, $\{x\}$, $\{y\}$, $\{z\}$, $\{1+1\}$, $\{1+2\}$, $\{1+x\}$, $\{1+y\}$, $\{1+z\}$, $\{2+1\}$, $\{2+2\}$, $\{2+x\}$, $\{2+y\}$, $\{2+z\}$, $\{x+1\}$, $\{x+2\}$, $\{x+x\}$, $\{x+y\}$, $\{x+z\}$, $\{y+1\}$, $\{y+2\}$, $\{y+x\}$, $\{y+y\}$, $\{y+z\}$, $\{z+1\}$, $\{z+2\}$, $\{z+x\}$, $\{z+y\}$, $\{z+z\}$

- **[2] : Transfer Point** $\Rightarrow x = 1$

  $\{1, x\}$, $\{2\}$, $\{y\}$, $\{z\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{1+2, x+2\}$, $\{1+y, x+y\}$, $\{1+z, x+z\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{2+y\}$, $\{2+z\}$, $\{y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$, $\{y+z\}$, $\{z+1, z+x\}$, $\{z+2\}$, $\{z+y\}$, $\{z+z\}$

- **[3] : Transfer Point** $\Rightarrow y = x + 2$

  $\{1, x\}$, $\{2\}$, $\{z\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{y, 1+2, x+2\}$, $\{1+z, x+z\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{2+z\}$, $\{z+1, z+x\}$, $\{z+2\}$, $\{z+z\}$, $\{1+y, x+y\}$, $\{2+y\}$, $\{y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$, $\{y+z\}$, $\{z+y\}$

- **[4] : Transfer Point** $\Rightarrow z = 1$

  $\{1, x, z\}$, $\{2\}$, $\{1+1, 1+x, 1+z, x+1, x+x, x+z, z+1, z+x, z+z\}$, $\{y, 1+2, x+2, z+2\}$, $\{2+1, 2+x, 2+z\}$, $\{2+2\}$, $\{1+y, x+y, z+y\}$, $\{2+y\}$, $\{y+1, y+x, y+z\}$, $\{y+2\}$, $\{y+y\}$

- **[5] : Transfer Point** $\Rightarrow z = 2$

  $\{1, x\}$, $\{2, z\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{y, 1+2, 1+z, x+2, x+z\}$, $\{2+1, 2+x, z+1, z+x\}$, $\{2+2, 2+z, z+2, z+z\}$, $\{1+y, x+y\}$, $\{2+y, z+y\}$, $\{y+1, y+x\}$, $\{y+2, y+z\}$, $\{y+y\}$

- **[6] : Confluence of** $\{4, 5\}$

  $\{1, x\}$, $\{2\}$, $\{1+1, 1+x, x+1, x+x\}$, $\{y, 1+2, x+2\}$, $\{2+1, 2+x\}$, $\{2+2\}$, $\{1+y, x+y\}$, $\{2+y\}$, $\{y+1, y+x\}$, $\{y+2\}$, $\{y+y\}$, $\{z\}$, $\{1+z, x+z\}$, $\{2+z\}$, $\{y+z\}$, $\{z+1, z+x\}$, $\{z+2\}$, $\{z+y\}$, $\{z+z\}$

- **[7] : END**

  $\{1, x\},\ \{2\},\ \{1+1, 1+x, x+1, x+x\},\ \{y, 1+2, x+2\},\ \{2+1, 2+x\},\ \{2+2\},$
  $\{1+y, x+y\},\ \{2+y\},\ \{y+1, y+x\},\ \{y+2\},\ \{y+y\},\ \{z\},\ \{1+z, x+z\},\ \{2+z\},$
  $\{y+z\},\ \{z+1, z+x\},\ \{z+2\},\ \{z+y\},\ \{z+z\}$

Chapter 6

Implementation Platform - Clang/LLVM

A modern compiler has three basic components - frontend, optimiser and backend. The frontend takes the high level source code and outputs the intermediate representation (IR) which serves as the input for the optimiser. The optimiser modifies the IR to make it more simpler and efficient - this new modified IR is then fed into the backend which gives the final low level code.

LLVM is a complete compiler infrastructure - a collection of libraries built to support compiler development and related tasks. Each library supports a particular component in a typical compiler pipeline - lexing, parsing, optimizations of a particular type, machine code generation for a particular architecture, etc. The central part of the LLVM project are **LLVM intermediate representation (LLVM IR)** and **LLVM core**. LLVM IR is the intermediate representation used in the LLVM project and LLVM core is responsible for all the optimisations and transformations that happens on the IR. The most important thing about the project is it provides all the facilities for anyone to write their own optimisations.

Clang is a compiler frontend for the C family of programming languages (`C, C++, Objective-C` etc.). It builds on the LLVM optimizer and code generator, allowing it to

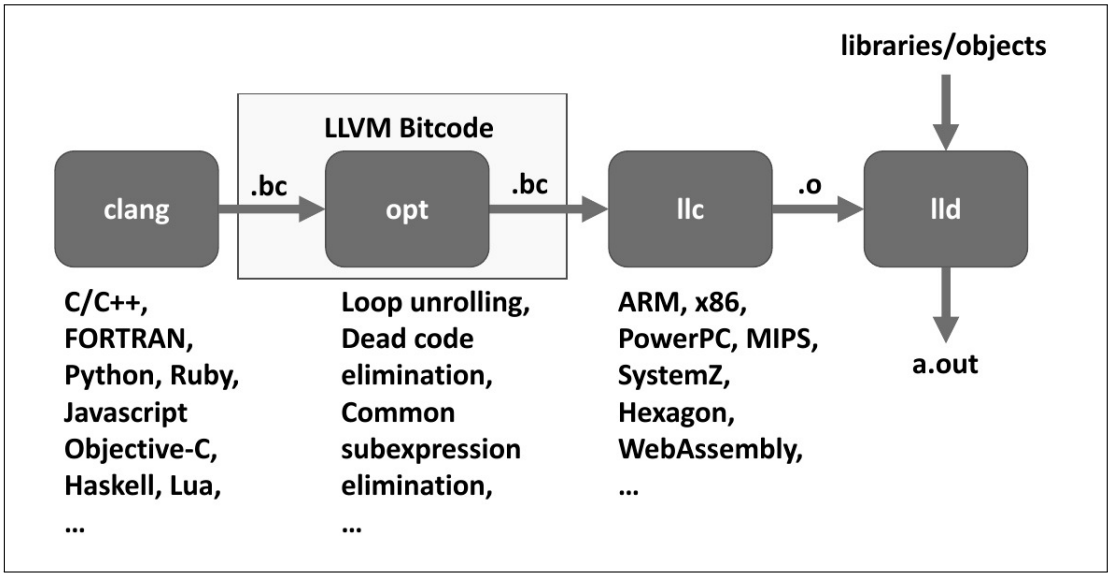

Figure 6.1 Various stages of compilation using Clang

provide high-quality optimization and code generation support for many targets.

6.1 Common Clang/LLVM Commands

In each of the following commands, an optional output file can be specified using `'-o'` flag.

- `'clang hello.c'` - Compiles *hello.c*.
- `'clang++ hello.cpp'` - Compiles *hello.cpp*.
- `'clang -emit-llvm -S hello.c'` - Gives LLVM IR corresponding to *hello.c* in text format file *hello.ll*.
- `'clang -emit-llvm -c hello.c'` - Gives LLVM IR corresponding to *hello.c* in binary bitcode format file *hello.bc*.
- `'llvm-as hello.ll'` - Converts *hello.ll* to *hello.bc*.
- `'llvm-dis hello.bc'` - Converts *hello.bc* to *hello.ll*.
- `'lli hello.bc'` - Directly executes *hello.bc*.
- `'lli hello.ll'` - Directly executes *hello.ll*.

NOTE - Sometimes Clang/LLVM version information is also required for running the commands - `'clang-8'`, `'clang++-8`, `'llvm-as-8'`, `'llvm-dis-8'`, `'lli-8'` etc.

6.2 Working with Clang/LLVM

This project is done on **Ubuntu 19.10** operating system using Clang version 8.0.1 and LLVM version 8.0.1. The instructions mentioned here are verified to work on the same.

6.2.1 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.

6.2.2 Building LLVM from source

Before proceeding make sure 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 extracted 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.

6.2.3 Writing a Pass

A **pass** is a program that takes an LLVM IR as input and transforms it generating a new IR. This new IR can be optimised version of the input or some modified version suitable for other tasks. Sometimes a pass only analyses the IR without modifying it.

This section explains how to create a simple pass named **HelloPass**.

- Create directory *'LLVMsrc/lib/Transforms/HelloPass'*. This directory will contain files related to the pass.
- Create a new file named *helloPass.cpp* inside *HelloPass* directory. This file will contain the code for the 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 the function-pass defined inside **helloPass** structure (the template arguments in the last line) is to be executed.

- 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 (without *.so extension) inside *'LLVMbuild/lib/'* directory, which will contain the pass when built. *helloPass.cpp* in the second line specifies the file which contains the source code for the 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 the files related to the pass and will also be used by **make** while building.

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

6.2.4 Running the pass

- Rebuild the LLVM source so that it includes the new pass that was added. For this change current directory to *LLVMbuild* and run `'make'`.
- Now run the pass on a binary bitcode file *source.bc*, containing an LLVM IR as `'./bin/opt -load ./lib/LLVMhelloPass.so -hello source.bc -o sourceN.bc'`. The pass can also be run on a `'*.ll'` file in a similar way.

NOTE - `'LLVMhelloPass'` is the same name that was specified in the *CMakeLists.txt* file of the pass folder. Also, `'-hello'` flag is the name by which the pass was registered.

Chapter 7

LLVM Implementation of the Algorithm

This chapter provides details of implementation of the Herbrand equivalence algorithm for Clang/LLVM compiler framework. The actual implementation along with other details like how to run and how to interpret the output etc. can be found on GitHub and for extensive documentation refer to LLVMDocs.

7.1 LLVM Intermediate Representation (IR)

Performing analysis and optimizations on a high level language code is very difficult. So compilers first convert the high level code to some intermediate representation. An IR is designed to be closer to assembly language yet independent of the target machine. It is designed such that the analysis and optimizations can be performed more conviniently compared to doing the same on the high level language. IRs are usually abstract internal representations rather than actual language.

LLVM IR is the intermediate form used by LLVM. Rather than just being an abstract internal representation, LLVM is a complete language on its own. It even provides human-readable assembly format for the IR. To get readable IR corresponding to file *hello.c*, run

'`clang-8 -emit-llvm -S hello.c -o hello.ll`'. The IR file can also be directly executed as '`lli hello.ll`'.

Apart from already existing optimizations in LLVM, one can write their own optimizations/analysis programs called **LLVM passes**. This chapter discusses the details of the pass written to perform Herbrand equivalence analysis. A brief tutorial on writing a basic pass is already covered in Section 6.2.

7.1.1 Basics of LLVM IR

- The temporaries and labels in LLVM IR has no name; however in **.ll* files they are named as `%0, %1, %2, ...`. But for reference names can be assigned to them.
- All the local variables in a function are allocated space on the stack using `alloca` instruction in the beginning of the function itself. Pointers to these locations are assigned to temporaries in order to allow access to them.
- Each time a value is assigned to be a local variable, it is updated immediately on the stack using its corresponding temporary with `store` instruction. Similarly each time a variable's value is to be accessed, it is loaded into a new temporary using `load` instruction.
- The result of every computation is stored in a new temporary and these temporaries can be assigned values only once. The values assigned to actual program variables (stored on stack) can be changed indirectly through their pointers; but the temporaries actually holding the pointers are also not reassignable.
- Example - Consider the instruction `x = y + z` and suppose that `%0, %1, %2` holds the stack locations allocated to `x, y, z` respectively. This single instruction would correspond to four instructions in the IR. First load the values of `y` and `z` in new temporaries `%3` and `%4`; `%5` is assigned `%3 + %4` and finally the value in `%5` is stored in location pointed by `%0`.
- Constants and temporaries are of type `Value` in LLVM and they can be accessed

through pointers (of type `Value*`). Instructions are of type `Instruction` which itself is derived from class `Value`. Functions have type `Function`.

NOTES

- In the implementation, for stack variables - the temporaries storing their addresses have been taken to represent the actual variables themselves. This way from the implementation point of view, such temporaries can change their values.
- The temporaries in the output are named as `T1, T2, ...`

7.2 Herbrand Equivalence Pass

7.2.1 Data Structures

- `ExpressionTy` - `std::tuple<char, Value*, Value*>` to represent an expression.
- `CfgNodeTy` - Represents a control flow graph node and has following data members -
  - `NodeTy` - `Enum` to denote the kind of node - `START`, `END`, `TRANSFER`, `CONFLUENCE`.
  - `instPtr` - `Instruction*` pointing to the instruction that defines the transfer function if the node corresponds to a transfer point.
  - `predecessors` - `std::vector<int>` containing indexes of predecessor control flow graph nodes (see `CFG`).

7.2.2 Global Variables

- `Constants` - `std::set<Value*>` storing the constants used in the program.
- `Variables` - `std::set<Value*>` storing the variables used in the program.
- `Ops` - `std::set<char>` storing the operands appearing in the programs on which the pass is run.
- `IndexExp` - `std::map<ExpressionTy, int>` to map program expressions of length atmost two to integers for indexing purpose.

- `SetCnt` - Next set identifier to be used.
- `Partitions` - `std::vector<std::vector<int>>` to keep track of partition at each program point. `Partitions[v][n]` stores the set identifier each expression `e` such that `IndexExp[e]` is `n`, at program point represented by `CFG[v]`. Also, `Partitions[v]` is the **partition vector** representing the partition at program point `CFG[v]` (see `CFG`).
- `Parent` - `std::map<std::tuple<char, int, int>, int>` to store parent set identfiers.
- `CFG` - `std::vector<CfgNodeTy>` to store program control flow graph.
- `CfgIndex` - `std::map<Instruction*, int>` to store the index of control flow graph node corresponding to instructions in the program, for which they define the transfer function (see `CFG`).

NOTE - For more details on `SetCnt`, `Partitions` and `Parent` refer to Section 4.1.

7.2.3 Functions

- `assignNames(F)` - Assign names to basic blocks and temporaries in function `F` for easy reference.
- `assignIndex(F)` - First iterates over instructions in function `F` to update `Constants` and `Variables`. Then updates `IndexExp` assigning integer indexes to expressions (of length atmost two) arbitrarily at the beginning of the analysis, for indexing purpose.
- `createCFG(F)` - Creates control flow graph corresponding to function `F` by updating `CFG`.
- `printValue(v)` - Function to print a constant or a variable (of type `Value*`) in a readable format.
- `printExpression(e)` - Function to print expression `e` (of type `ExpressionTy`) in a readable format.
- `printCode(F)` - Prints function `F` in a readable format.
- `printCFG()` - Prints control flow graph in readable format (see `CFG`).

- `printPartition(p)` - Prints partition vector `p` (of type `std::vector<int>`) in a readable format.
- `samePartition(p1, p2)` - Checks if the two partition vectors (`std::vector<int>`) `p1` and `p2` are same. They are same when values (ie. set identifiers) at two different indexes are equal in the first iff they are so in the second.
- `findSet(p, e)` - Finds set identifier representing expression `e` (which is of type `ExpressionTy`) in partition vector `p`.
- `findInitialPartition(p)` - Intialises partition vector `p` with $\perp$.
- `getClass(p, n, c)` - Updates `c` (`std::set<int>`) to contain indexes of expressions which are equivalent to expression with index `n`, in the partition vector `p`.
- `transferFunction(n)` - Applies corresponding transfer function to the partition at program point `CFG[n]`.
- `confluenceFunction(n)` - Applies the confluence operation to the partition at program point `CFG[n]`.
- `HerbrandAnanlysis(F)` - Performs Herbrand analysis over function `F`.

NOTE - For more details on the function implementation, refer to Chapter 4 and for extensive documentation refer to LLVMDocs.

7.3 Benchmarking

The idea was to use the analysis information to perform program optimizations - like constant propagation, constant folding, common subexpression elimination etc. These optimizations were to be benchmarked using SPEC benchmarking suites. The SPEC establishes, maintains and endorses standardized benchmarks and tools to evaluate performance and energy efficiency for the newest generation of computing systems. In particular **SPEC CPU benchmarks** are used for measuring and comparing compute intensive performance, stressing a system's processor, memory subsystem and *compilers*. Here, the CPU bench-

marks were to be used for comparing the performance of our optimizations with respect to ones already existing like common subexpression elimination, global value numbering both in terms of processing time (time taken to optimize a program) and execution time (of the optimized program).

The SPEC test cases are actual real world programs with slight modifications focussing on the kind of benchmarks to be done; some of these includes GCC, Perl interpreter, video compression program etc. These are large programs and for an optimization pass to be successfully benchmarked, it must be able to correctly handle all the constructs in LLVM IR. One must know how the high level language constructs like different basic types, pointers, classes, unions, enumerations etc. are translated in the IR; how to operate with them in the IR; and what are the feasible modifications that can be peformed without breaking the consistency and correctness of the programs. In short, one must have very deep and extensive knowledge of LLVM. This is very difficult given the size of the LLVM project itself and limited tutorials available and even the ones available are very basic. Though the documentation is extensive, it is useful only as a reference and is of not much use to a know-nothing.

The current implementation has many limitations. It considers only integer variables and doesn't expect other basic types like floats, boolean, characters etc. Also it can't handle derived and complex types like structures, classes, unions, enumerations, pointers etc. It considers only five arithematic operators - `+, -, *, /, %`.

Though some of these can be easily handled, there are still others and the ones not mentioned like handling global variables, recursion, dynamic memory allocation etc. Given these complications it was decided to leave the implementation as such and skip the benchmarkings.

Chapter 8

Toy Language Implementation of the Algorithm

This chapter provides details of `C++` implementation of the Herbrand equivalence algorithm for a custom toy language. The actual implementation along with other details like how to run and how to interpret the output etc. can be found on GitHub and for extensive documentation refer to ToyLanguageDocs.

8.1 The Toy Language

There are three different kinds of statements in a toy language program.

- **Normal Instructions**

  These are of one of the following three forms where `x` should be a variable, `y` and `z` can be either an integer or a variable. The last one specifies **non-deterministic assignment**.

  - `x = y`
  - `x = y + z`
  - `x = *`

Variables do not require declaration for use.

- **`LABEL` Statements**
  - Such statements define labels which act as alias for their immediate next statement of the **first kind** (ie. the normal instructions). A `LABEL` statement occurring at the end refers to the **end of the program**.
  - It has following format - `'LABEL identifier1 identifier2 ...'`.
  - A normal instruction can either have a single label or multiple labels. Multiple labels can either be defined in the same `LABEL` statement or different statements.
  - All the `LABEL` identifiers used in the definitions must be unique (even those used for the same instruction).
- **`GOTO` Statements**
  - `GOTO` statements are used to specify jumps - the successors of immediately preceding instruction of the **first kind** (ie. the normal instructions). A `GOTO` statement at the beginning means two different control flow paths from the **start of the program**.
  - It has following format - `'GOTO identifier1 identifier2 ...'`.
  - Again like `LABEL` statements, for specifying multiple successors of a normal instruction there can either be a single `GOTO` statement with multiple label identifiers or multiple `GOTO` statements.
  - For Herbrand equivalence analysis the condition of the jumps are irrelavent, so in the toy language all the jumps are unconditional.
  - For instructions of the **first kind** without `GOTO`, by default their successor is the immediate occurring instruction of the **first kind**. However, if such instructions has a corresponding `GOTO`, all the successors **must** be specified there itself - they do not have any default successors.
  - The labels used in the `GOTO` statement must be defined in some `LABEL` statement in the program.

Other important points

- Each instruction must be specified a single line and should be the only instruction in that line.
- Empty lines in the program text are ignored.
- A normal instruction can contain arithmatic operators other than `+`.
- A variable name must start with an alphabetic character.
- Specify the program properly with atleast one whitespace between the tokens. Also check whether the program is parsed properly by looking at the output when the program is executed.
- Variables and constants defined in any unreachable instruction are also considered for analysis, but the instruction itself is ignored.

8.2 `MapVector.h`

This file defines a `MapVector` data structure for bi-directional mapping to integers. Internally it uses `std::map` for forward mapping to integers and `std::vector` for reverse mapping. The data structure is generalised using `templates`.

8.2.1 Private Data Members

- `Map` - Internal `std::map` variable for forward mapping.
- `Vector` - Internal `std::vector` variable for reverse mapping.

8.2.2 Public Member Functions

- `MapVector()` - Default constructor.
- `begin()` - Returns forward iterator to the beginning of `Vector`.
- `end()` - Returns forward iterator to the end of `Vector`.
- `rbegin()` - Returns reverse iterator to the end of `Vector`.

- `rend()` - Returns reverse iterator to the beginning of `Vector`.
- `empty()` - Returns boolean to indicate whether the `MapVector` object is empty.
- `size()` - Returns the number of unique items inserted into `MapVector` object.
- `insert(item)` - Inserts `item` in the `MapVector` object if not already present.
- `getInt(item)` - Returns the integer to which `item` is forward mapped.
- `operator[](n)` - Returns `item` which is forward mapped to integer n.
- `clear()` - Clears the `MapVector` object by deleting existing items.

8.2.3 How it works

When an `item` is to be inserted in a `MapVector` object, first a check is made if the `item` is already present. If not found, it is inserted into the internal `Map` data member, being forward mapped to next available integer x starting from 0 (which is same as `Vector.size()` or `Map.size()` before insertion). Also at the same time, it is inserted into the `Vector` data member at the same integer index x for reverse mapping.
So, `MapVector` provides $\mathcal{O}(\log n)$ forward mapping and $\mathcal{O}(1)$ reverse mapping, where n is the size of the `MapVector` object. Insertion operation is also $\mathcal{O}(\log n)$.

8.3 `Program.h`

This file defines a `Program` data structure for capturing a toy language program. In addition it defines a `CustomOStream` class whose object acts as a placeholder inside `Program` for providing `std::cout` like functionality for `Program` class.
The details of `CustomOStream` class are irrelavent and hence not mentioned. The following subsections provide details of `Program` class.

8.3.1 Public Data Structures

- `ValueTy` - Represents constants and variables in the program. It has following data members -

  - `isConst` - Boolean to indicate whether the corresponding object represents a constant or a variable.
  - `index` - Corresponding integer index for the constant or variable (see `Constants` and `Variables` data members of `Program` class).

- `ExpressionTy` - Represents an expression. It has following data members -
  - `op` - Character to represent the operand in the expression.
  - `leftOp` - `ValueTy` object to represent the left operand.
  - `rightOp` - `ValueTy` object to represent the right operand.

- `InstructionTy` - Represents a normal instruction in the program. It has following data members -
  - `lValue` - `ValueTy` object for lvalue of the instruction.
  - `rValue` - `ExpressionTy` object for rvalue of the instruction.
  - `reachable` - Boolean to indicate whether the instruction is reachable.
  - `cfgIndex` - Index of the node in control flow graph for which the instruction defines the transfer function (see `CFG` data member of `Program` class).
  - `predecessors` - `std::set<int>` of indexes of predecessors of the instruction (see `Instructions` data member of `Program` class).

- `CfgNodeTy` - Represents a control flow graph node corresponding to the program. It has following data members -
  - `predecessors` - `std::vector<int>` object of indexes of predecessor control flow graph nodes (see `CFG` data member of `Program` class).
  - `instructionIndex` - Index of the instruction that defines the transfer function if the node represents a transfer point (see `Instructions` data member of `Program` class).

8.3.2 Public Data Members

- `Variables` - `MapVector<string>` to store program variables.
- `Constants` - `MapVector<int>` to store program constants.
- `Instructions` - `std::vector<InstructionTy>` object to store program instructions.
- `CFG` - `std::vector<CfgNodeTy>` to store program control flow graph.
- `cout` - `CustomOStream` object to provide `std::cout` like functionality for `Program` class.

8.3.3 Public Member Functions

- `parse(filename)` - Parses toy language program in `filename` and updates `Variables`, `Constants` and `Instructions` data members.

  It performs two passes over the program. In the first pass, it stores the instructions, performs basic checks to ensure syntactic correctness, determines labels and jumps. In the second pass it performs BFS from the first instruction, mapping jumps to actual instructions and also determining the reachability of instructions. The function `throw`s an error in case any of the checks fails.

- `print()` - Prints the parsed program in readable format.

- `createCFG()` - Creates control flow graph corresponding to the parsed program by updating `CFG` data members. The control flow graphs contains nodes corresponding to transfer points, confluence points, `START` and `END`. Unreachable instructions are ignored while creating the control flow graph.

  It also performs two passes over the program. In the first pass it assigns indexes to the transfer points (reachable instructions) and confluence points in `CFG` data member. And then in the second pass, it actually fills the `CFG` vector.

- `printCFG()` - Prints the control flow graph in readable format.

NOTES

- `Program.h` also overloads `operator<` for `ValueTy` and `ExpressionTy` so that `std::set` and `std::map` objects can be created using them.
- For `std::cout` like facilities, `operator<<` is also overloaded for `CustomOStream` to output `char`, `int`, `string`, `ValueTy`, `ExpressionTy`, `InstructionTy` objects.

8.4 `HerbrandEquivalence.cpp`

This file implements the Hebrand equivalence algorithm for the toy language described above.

8.4.1 Global Variables

- `program` - `Program` object to store parsed toy program.
- `Ops` - `std::vector<char>` object to store the operators that can occur in the toy program being analysed.
- `IndexExp` - `std::map<Program::ExpressionTy, int>` to map program expressions of length atmost two to integers for indexing purpose.
- `SetCnt` - Next set identifier to be used.
- `Partitions` - `std::vector<std::vector<int>>` to keep track of partition at each program point. `Partitions[v][n]` stores set identifier for expression `e` such that `IndexExp[e]` is `n`, at program point represented by `program.CFG[v]`. `Partitions[v]` is the **partition vector** representing the partition at program point `program.CFG[v]`.
- `Parent` - `std::map<std::tuple<char, int, int>, int>` to store parent set identifiers.

NOTE - For more details on `SetCnt`, `Partitions` and `Parent` refer to Section 4.1.

8.4.2 Functions

- `assignIndex()` - Updates `IndexExp` assigning integer indexes to expressions arbitrarily at the beginning of the analysis, for indexing purpose.
- `samePartition(p1, p2)` - Checks if the two partition vectors (`std::vector<int>`) `p1` and `p2` are same. They are same when values (ie. set identifiers) at two different indexes are equal in the first iff they are so in the second.
- `findSet(p, e)` - Finds set identifier representing expression `e` (which is of type `Program::ExpressionTy`) in partition vector `p`.
- `findInitialPartition(p)` - Intialises partition vector `p` with $\perp$.
- `getClass(p, n, c)` - Updates `c` (`std::set<int>`) to contain indexes of expressions which are equivalent to expression with index `n`, in the partition vector `p`.
- `printPartition(p)` - Prints partition vector `p` in readable format.
- `transferFunction(n)` - Applies corresponding transfer function to the partition at program point `program.CFG[n]`.
- `confluenceFunction(n)` - Applies the confluence operation to the partition at program point `program.CFG[n]`.
- `HerbrandEquivalence()` - Main Herbrand analysis function.
- `main()` - Parses toy program (`program.parse(argv[1])`), creates control flow graph (`program.createCFG()`) and then calls `HerbrandEquivalence()` function.

NOTE - For more details on the function implementation, refer to Chapter 4 and for extensive documentation refer to ToyLanguageDocs.

Chapter 9

Proof Attempt

Section 9.1 contains details of the attempt made to prove the correctness of the algorithm - reason for the mentioned approach, claims made and reasons for its failure. Section 9.2 informally shows why the algorithm should work for the special case when the input program has no loops.

9.1 Augmented Program Approach

Let $\mathcal{X}$ and $\mathcal{I}$ be the set of variables and instructions in the program respectively. Also, let $\mathcal{T}$ and V be the set of terms and program points; and $\mathcal{D}(V, \mathcal{F})$ be the corresponding data flow framework. Let $\mathcal{W} \subseteq \mathcal{T}$ be the corresponding set of terms of length atmost two (in terms of number of operands).

The basic motive behind this approach was to prevent the disappearance of some equivalence classes in between the program due to reassignment to variables (see Section 5.4 for an example). The classes disappear because there exists no expression in our working set $\mathcal{W}$ that belongs to that class however that class still contains expressions outside $\mathcal{W}$.

9.1.1 Augmented Program

Every instruction in the original program is extended to a **bundle** of instructions - for each instruction $i \in \mathcal{I}$, add immediately after it dummy instructions of the form $x_i \leftarrow x, \forall x \in \mathcal{X}$. This new program with dummy instructions is called the **augmented program** and the new variables introduced are called **shadow variables**.

Let $\overline{\mathcal{X}}$, $\overline{\mathcal{I}}$, $\overline{\mathcal{T}}$ and $\overline{V}$ be the corresponding sets of variables, instructions, terms and program points respectively; and $\overline{\mathcal{D}}(\overline{V}, \overline{\mathcal{F}})$ be the corresponding data flow framework. Also $\overline{\mathcal{W}} \subseteq \overline{\mathcal{T}}$ be the corresponding set of terms restricted to length atmost two.

9.1.2 Idea

Let $\mathcal{L}$ and $\overline{\mathcal{L}}$ be the infinite lattices associated with the original and augmented programs respectively. These are same as $\overline{\mathcal{G}(\mathcal{T})}$ described in Section 3.2, for the original and augmented programs.

Given a partition $\mathcal{P}$, define

$$\Phi_W(\mathcal{P}) = \{A_i \cap W \mid A_i \in \mathcal{P} \text{ and } (A_i \cap W) \neq \phi\}$$

Let Φ_W be generalised so that when applied to set of partitions $\mathcal{S}$,

$$\Phi_W(\mathcal{S}) = \{\Phi_W(\mathcal{P}) \mid \mathcal{P} \in \mathcal{S}\}$$

It is clear that $\mathcal{L} = \Phi_{\mathcal{T}}(\overline{\mathcal{L}})$. Let $\mathcal{L}_{\mathcal{W}} = \Phi_{\mathcal{W}}(\mathcal{L})$ and $\overline{\mathcal{L}}_{\overline{\mathcal{W}}} = \Phi_{\overline{\mathcal{W}}}(\overline{\mathcal{L}})$ be the finite lattices associated with the original and augmented lattices respectively.

Let $f_{\mathcal{D}}$ and $\overline{f}_{\overline{\mathcal{D}}}$ be be the composite transfer function associated $\mathcal{L}$ and $\overline{\mathcal{L}}$ respectively, as defined in Chapter 3. The Hebrand congruence functions are given by

$$\mathcal{H}_{\mathcal{D}} = \bigwedge\{\top, f_{\mathcal{D}}(\top), f^2_{\mathcal{D}}(\top), ...\} \text{ and } \overline{\mathcal{H}}_{\overline{\mathcal{D}}} = \bigwedge\{\overline{\top}, \overline{f}_{\overline{\mathcal{D}}}(\overline{\top}), \overline{f}^2_{\overline{\mathcal{D}}}(\overline{\top}), ...\}$$

for the original and augmented lattice. Here, $\top$ and $\overline{\top}$ stands for top element in the product lattices corresponding to $\mathcal{L}$ and $\overline{\mathcal{L}}$ respectively.

Observations

- $\Phi_{\mathcal{W}}(\mathcal{H}_{\mathcal{D}}) = \Phi_{\mathcal{W}}(\overline{\mathcal{H}}_{\overline{\mathcal{D}}})$
- $\Phi_{\mathcal{W}}(\Phi_{\overline{\mathcal{W}}}(\overline{\mathcal{H}}_{\overline{\mathcal{D}}})) = \Phi_{\mathcal{W}}(\mathcal{H}_{\mathcal{D}})$

Claim

$$\Phi_{\overline{W}}(\overline{\mathcal{H}}_{\overline{\mathcal{D}}}) = \text{ Maximum fix point of transfer function } \overline{f}_{\overline{\mathcal{W}}} \text{ on } \overline{\mathcal{L}}_{\overline{\mathcal{W}}}$$

The right hand side can computed efficiently using the algorithms in Chapter 4, on the augmented program.

9.1.3 Hypothesis

Following hypothesis was proposed which was supposed to be used later to prove the claim - suppose at program point $v \in V$ in the original program, variable $x \cong t$ (where $\mid t \mid \geq 2$) in our original infinite lattice $\mathcal{L}$, then at the output of the bundle corresponding to program point v in the augmented program, there exists t' in augmented finite lattice $\overline{\mathcal{L}}_{\overline{\mathcal{W}}}$ with $\mid t' \mid = 2$ which is equivalent to x and consists only of constants and shadow variables.
However, a counter example to the hypothesis was discovered while attempting the proof for the hypothesis. Consider test case in Section 5.13 - at the confluence point (program point 6), in the original infinite lattice x and $((y + z) + n)$ are equivalent; but in the augmented finite lattice no two length expression consisting of only shadow variables and constants that is equivalent to x.

9.2 Why the Algorithm Works

Instead of a general case, consider the ones in which the program has no loops. In such cases the control flow graph (CFG) is a directed acyclic graph (DAG). And if the algorithm processes CFG nodes in the topological order, it converges in the first iteration itself.

The `Parent` map captures the syntactic information present in the expressions in the form of an ordered tree. Each node in the tree is assigned a unique set identifier and represents some syntactically equivalent expressions. Suppose a node has set identifier x and `Partitions[v][e]` is x - it means that at program point v, expression e has same syntactic value (across all program paths from the start) as represented by the node. A node captures this syntactic information in the form of its two ordered childrens and an operator - which is precisely the information stored the `Parent` map. Leaf nodes represents atomic values - a constant, non-deterministic or variable (variable in terms of different values along different execution paths) value. These properties implies that two expressions that are assigned the same set identifier at a program point are Herbrand equivalent at that point, which shows the correctness of the algorithm.

Now induction can be used to show that the algorithm actually maintains the properties mentioned in the last paragraph when it processes the nodes in the topological order.
At the `START` point all the expressions in $\mathcal{W}$ are inequivalent to each other. And the algorithm also assigns different set identifiers to elements of $\mathcal{W}$. Also, for all $x_1 + x_2 \in (\mathcal{W} \backslash (\mathcal{C} \cup \mathcal{X}))$, `Parent[{+, Partitions[START][`$x_1$`], Partitions[START][`$x_2$`]}]` is assigned `Partitions[START][`$x_1 + x_2$`]`. So the properties hold at `START` point.

Now the properties will be shown to hold for program point v, assuming that they hold at all points which occur before v in the topological order - in particular they hold for all $u \in$ `Predecessors[v]`.

- **v is a transfer point**

  Suppose v represents the assignment $x = e$ and u is its predecessor. With respect to u,

the only expressions changing values are the ones which contain x. So `Partitions[u]` is copied to `Partitions[v]` and x is assigned the set identifier of e because they now have equal values.

Now consider all expressions of the form $a + b$ which contains x. If `Parent[{+, Partitions[v][a], Partitions[v][b]}]` is defined it means that there already exists a node that represents expressions syntactically equivalent to the value of $a + b$ at the current point. So `Partitions[v][a + b]` should be assigned the same set identifier. However, if no such node exists a new one needs to be created and is to be assigned to `Parent[{+, Partitions[v][a], Partitions[v][b]}]` as well as to `Partitions[v][a + b]` to maintain the properties. This is consistent with what the algorithm actually does.

- v **is a confluence point**

  If $e \in \mathcal{W}$ has same set identifiers for all $u \in$ `Predecessors[v]`, it means that it has same syntactic value at all its predecssors and hence even at the confluence point - so it must be assigned the same set identifier (as that of the predecessors) even at v to maintain the properties.

  However, if $e \in \mathcal{W}$ has different identifiers at the predecessors then not all the values at the predecessor points are syntactically equivalent - for this case we require a new leaf node (because of variable values at predecessors). So we must find expressions which are equivalent to e at all the predecessors - and assign a new set identifier (new leaf node) for them to maintain the properties.

  These both cases are consistent with the algorithm.

Chapter 10

Conclusion

Detection of equivalent expressions in a program can be used in multiply ways - compilers can use it for program optimisations involving redundant expression elimination, dead code elimination etc.; in program verification and plagiarism detection tools etc. So Herbrand equivalence being a form of equivalent expression analysis has a huge importance both from theoretical as well as practical point of view.

10.1 Summary of Work Done

- Pointed out few errors in the original alogrithm proposed by Babu et al. [4] and refined it making it more efficient than the original version (see Section 4.3). The new updated algorithm is given in Chapter 4.
- Completed LLVM implementation (see Chapter 7) and the code for the same can be found on GitHub. Benchmarking was skipped as it is not feasible (see Section 7.3).
- In addition to LLVM implementation an additional implementation for a toy language is done (see Chapter 8) and its code can also be found on GitHub.
- An attempt for proving the algorithm (see Chapter 9) was made but it failed because counterexample to a hypothesis was found that was supposed to be used later in the

final proof (see Subsection 9.1.3). However an informal proof of why the algorithm should work is given in Section 9.2, for special case when the input program has no loops.

10.2 Future Scope

Though the algorithm gives correct results for the test cases on which it was tried, work for complete formal proof of its correctness is still ongoing. The algorithm is already efficient being polynomial in time with respect to the program size; so once the correctness is proven it can easily be employed for different applications mentioned earlier.

External References

- **GitHub repository for the project**

  github.com/himanshu520/HerbrandEquivalence

- **Documentation for LLVM implementation**

  himanshu520.github.io/HerbrandEquivalenceLLVMDocs/

- **Documentation for toy language implementation**

  himanshu520.github.io/HerbrandEquivalenceToyDocs/

- **LLVM download page**

  releases.llvm.org/download.html

- **Source code for LLVM-8.0.1**

  github.com/llvm/llvm-project/releases/download/llvmorg-8.0.1/

  llvm-8.0.1.src.tar.xz

- **LLVM language reference manual**

  llvm.org/docs/LangRef.html

- **Tutorial for writing an LLVM pass**

  llvm.org/docs/WritingAnLLVMPass.html

Bibliography

[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] 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.

[6] 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.

[7] 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.

[8] 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.

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