

# Program Analysis - Herbrand Equivalence

## MidTerm BTP Presentation

Himanshu Rai

Indian Institute of Technology, Palakkad

24/09/2019

# Optimizing Compilers

- Modern compilers perform a lot of optimizations on the code
- One common optimization performed is detecting equivalence of program subexpressions
- It is the main theme of many algorithms like constant propagation, constant folding, common subexpression elimination etc.
- Each of these detect a restricted class of expression equivalence
- In fact checking equivalence of program subexpressions have been shown to be an undecidable problem
- Informally, this means we can't have an algorithm to find all equivalent subexpressions in a program

# Herbrand Equivalence

- So, usually a restricted form of expression equivalence called *Herbrand Equivalence* is targeted
- Two expressions are Herbrand equivalent at a program point if they have syntactically the same value across all the execution paths from the start of the program to that particular point
- The operators are treated as uninterpreted functions

# Example of Herbrand Equivalence Computation

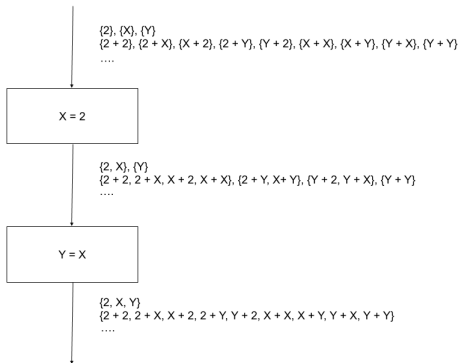


Figure: Example of Herbrand Equivalence at a *transfer point*

# Example of Herbrand Equivalence Computation

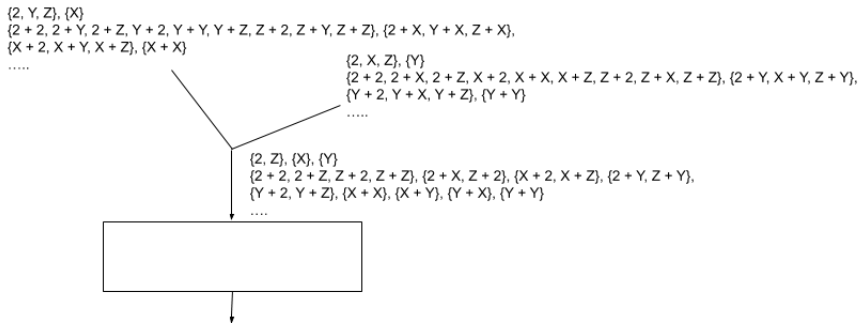
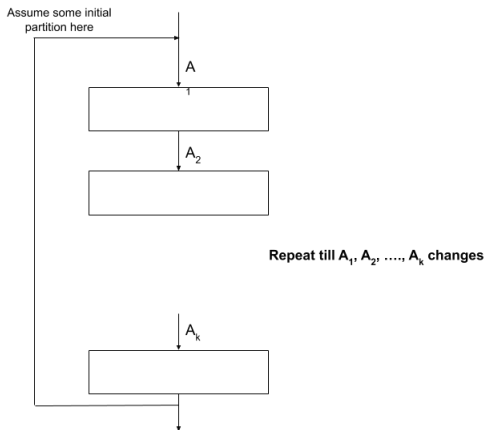


Figure: Example of Herbrand Equivalence at a *confluence point*

# Example of Herbrand Equivalence Computation



**Figure:** Example of Herbrand Equivalence in presence of a loop in the program graph

# Important points

Herbrand equivalence captures only syntactic equivalence, and not semantic

- $2 + 2$  is not equivalent to  $4$ , they are semantically equivalent and not syntactically
- $X + Y$  is not equivalent to  $Y + X$ , unless  $X$  and  $Y$  are equivalent. Because operators are treated as uninterpreted functions, so we cannot consider  $+$  to be distributive

- There have been several attempts at getting algorithms for computing Herbrand Equivalences
- Kildall (in 1973) gave an algorithm to find the Herbrand Equivalence classes. Kildall's algorithm is precise in the sense it finds all the Herbrand Equivalences but is exponential in time and space
- Alpern, Wegman and Zadek (in 1998) gave a polynomial time algorithm (AWZ) but it was not able to discover all the Equivalences
- Ruthing, Knoop and Steffen (in 1999) improved the AWZ algorithm. It was able to detect more equivalences but was still incomplete





- Gulwani and Necula (in 2007) gave an algorithm for finding Herbrand equivalence classes restricted to the program expression. This algorithm was considered complete but later Saleena and Paleri showed that Gulwani's algorithm is not complete for a related problem of global value numbering
- The problem is that most of these algorithms were based on fix point computations. But the classical definition of Herbrand equivalence is not a fix point based definition making it difficult to prove their precision or completeness

# Idea of the Project

- Babu, Krishnan and Paleri (in 2019) gave a new lattice theoretic formulation of Herbrand equivalences and proved its equivalence to the classical version
- Based on their theory they also gave an algorithm to compute *Herbrand equivalences associated with program expressions*
- However, the algorithm does not completely follows the theoretical work, so its correctness/precision needs to be established
- The task of this project is to implement the algorithm using LLVM framework and benchmark it against the standards
- Also, a proof of correctness/precision of the algorithm has to be presented

- Read two papers - by Gulwani and Necula[1]; and by Babu, Krishnan and Paleri[2]
- Started experimenting with LLVM, by writing some basic optimization pass

-  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.
-  J. Babu, K. M. Krishnan, and V. Palleri, “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.