

Program Analysis - Herbrand Equivalence

EndSem BTP Presentation

Himanshu Rai

Indian Institute of Technology, Palakkad

13/11/2019

- Detection and elimination of redundant expressions in a program is done by every industry level compiler
- This involves finding equivalence between subexpressions in the program - but it has been shown to be an undecidable problem
- This means we can't have an algorithm for it. So, compilers target some restricted form of equivalence. One such class of expression equivalence is **Herbrand Equivalence**

Herbrand Equivalence

- 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
- 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 commutative
 - Similarly, $X + (Y + Z)$ and $(X + Y) + Z$ are not equivalent, because we can't take $+$ to be associative. So, $X + Y + Z$ makes no sense.

Idea of the project

- There have been several attempts at getting algorithms for computing Herbrand Equivalences
- The algorithms given are either precise but exponential or polynomial but imprecise
- The problem is that most of these algorithms 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

Idea of the project

- Babu, Krishnan and Paleri [1] gave a new lattice theoretic formulation of Herbrand equivalences and proved its equivalence to the classical version
- Based on their theory they have given an algorithm to compute *Herbrand equivalences associated with program expressions*
- The algorithm deviates from theory, in the sense that instead of keeping track of all expressions that can be formed using constants and variables in the program, it restricts to expressions having length at most two
- The main idea of the project is to implement this algorithm for LLVM compiler framework

- **Till midterm** :- Read paper [1] to understand the problem, its theoretical background and the algorithm to implement. Papers [2, 3] contained similar works
- **After midterm** :- Finished first working implementation of the algorithm, for LLVM compiler framework. The code can be found [here](#)
- **Next** :-
 - Optimize the initial implementation for better performance
 - Currently equivalence information is only computed, using it perform optimizations
 - Benchmarking the implementation
 - Proving the correctness of the algorithm because it deviates from theory

Herbrand Equivalence Computation

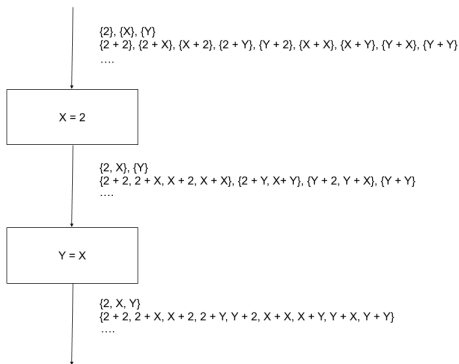


Figure: Example of Herbrand Equivalence computation at a **transfer point**

Herbrand Equivalence Computation

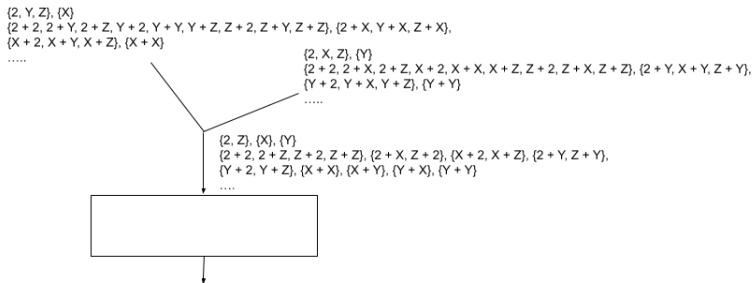


Figure: Example of Herbrand Equivalence computation at a **confluence point**

Herbrand Equivalence Computation

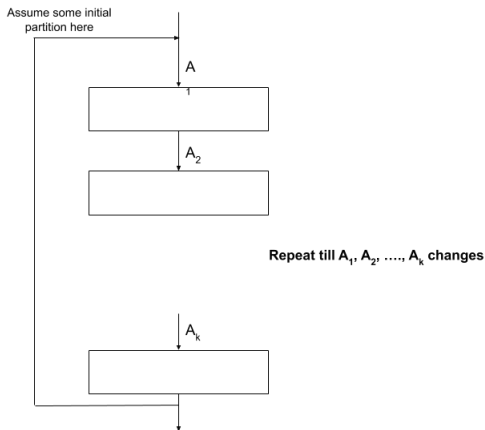





Figure: Example of Herbrand Equivalence computation in presence of a loop in the program graph

Learnings

- Working with LLVM

Challenges faced

- Working with LLVM - there is documentation, but difficult to understand for someone who is new to it
- Testing and debugging

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