Program Analysis - Herbrand Equivalence

MidTerm BTP Presentation

Himanshu Rai

Indian Institute of Technology, Palakkad

27/02/2020

Herbrand Equivalence

- **Herbrand Equivalence** is a kind of expression equivalence
- Informally, 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
  - If $Z = X$ then $Z \cong X$
  - If $Z = X + Y$ then $Z \cong X + Y$
  - $X \cong Y$ iff $X + Z \cong Y + Z$
  - $2 + 2 \ncong 4$
- The operators are treated as **uninterpreted functions**
  - $X + Y \ncong Y + X$
  - $X + (Y + Z) \ncong (X + Y) + Z$

Herbrand Equivalence Analysis

- In **Herbrand equivalence analysis**, our universe $\mathcal{U}$ is the set of **all possible expressions** that can be formed using constants, variables and operators in the program
- Then for **each program point**, we partition $\mathcal{U}$ such that two expressions belong to the same class iff they are Herbrand equivalent at that point
- Finding **semantic equivalence** of program expressions is **an undecidable problem** - so usually some **restricted form of equivalence** is considered - and Herbrand equivalence is one such

Example of Herbrand Equivalence Computation

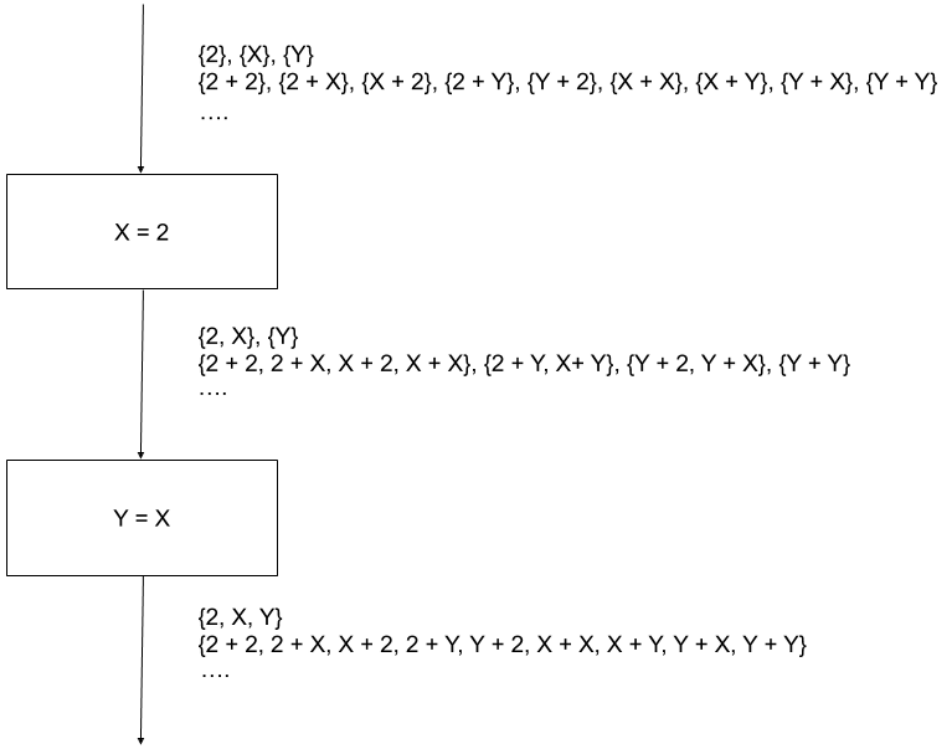

Figure: Example of Herbrand Equivalence at a **transfer point**

Example of Herbrand Equivalence Computation

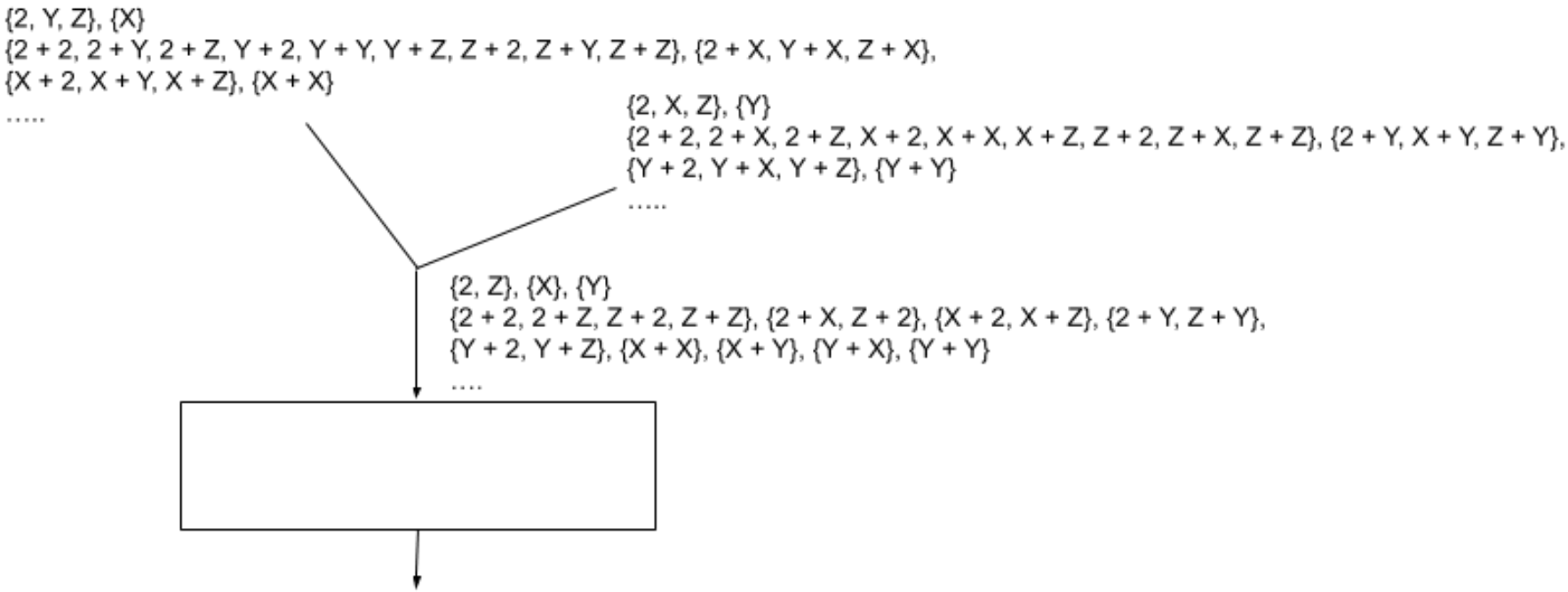

Figure: Example of Herbrand Equivalence at a **confluence point**

Idea of the Project

- Babu, Krishnan and Paleri (in 2019) has given a new lattice theoretic formulation of Herbrand equivalences and proved its equivalence to the classical version
- They also gave an algorithm to compute **Herbrand equivalences associated with program expressions** (expressions that can actually occur in a program)
- However, the algorithm does not completely follows the theoretical work, so its correctness/precision needs to be established

Idea of the Project

- The task of this project is to **implement the algorithm** for Clang/LLVM compiler framework
- Use the equivalence information to **perform optimizations** and benchmarking it
- Also, a **proof of correctness/precision** of the algorithm has to be presented

Work Done - Last Semester

- Read few papers related to past works, to get familiarity with the topic
- Finished initial implementation of the algorithm for Clang/LLVM compiler framework

Work Done - This Semester

- Added test cases for verification, corrected errors in the initial implementation
- Added Doxygen style comments for proper inline documentation
- Expanded code to perform optimizations using analysis information from the earlier implementations
- All the codes are available on GitHub and any further details can be found in the report

Optimizations done

- Three kinds of optimizations are done -
  - **Constant propagation** - If $X = 2$, then we can replace all uses of X by 2
  - **Constant folding** - Compute a constant expression at compile time rather than at runtime, eg. $X = 2 + 2$ is same as $X = 4$
  - **Redundant expression elimination** - If $X + Y$ is already computed, then we don't need to compute it again
- The optimization results are same as those by already existing GVN pass in LLVM, on the test cases it was run

Work in Progress

- We have already started working on the proof - we realised that modifying the algorithm a bit might be convinient for proving the correctness
- We are also parallelly working on getting the benchmarks done

Timeline

- Adding testcases, verification, improving code, documentation - 2 weeks
- Adding optimizations and verifying the results - 2 weeks
- Looking details for benchmarking - 1 week
- New codes and attempt for proof - 3 weeks