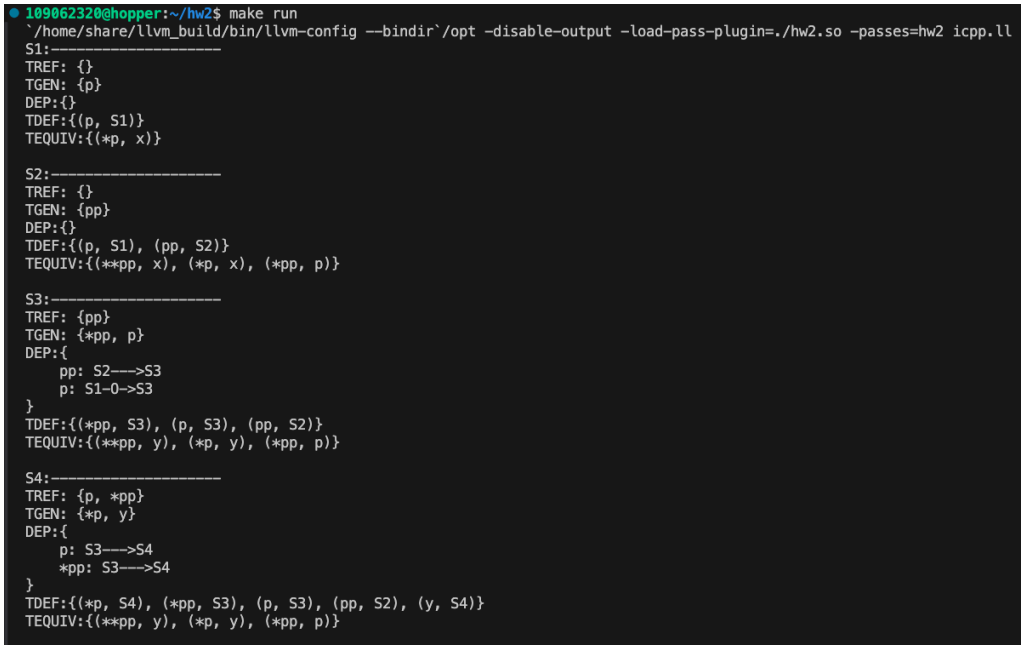
hw2_109062320_report

How to run your program (e.g. any paths that TA should modify in your Makefile)

All parts of my makefile are identical to the sample makefile in the specification, except for the path to llvm_build.

Display the output of the released testcases that you can handle

icpp



icpp2

```
109062320@hopper:~/hw2$ make && make run
make: Nothing to be done for 'all'.
`/home/share/llvm_build/bin/llvm-config --bindir`/opt -disable-output -load-pass-plugin=./hw2.so -passes=hw2 icpp2.ll
S1:--------------------
TREF: {}
TGEN: {p}
DEP:{}
TDEF:{(p, S1)}
TEQUIV:{(*p, x)}

S2:--------------------
TREF: {}
TGEN: {pp}
DEP:{}
TDEF:{(p, S1), (pp, S2)}
TEQUIV:{(**pp, x), (*p, x), (*pp, p)}

S3:--------------------
TREF: {pp}
TGEN: {*pp, p}
DEP:{
    pp: S2--->S3
    p: S1-O->S3
}
TDEF:{(*pp, S3), (p, S3), (pp, S2)}
TEQUIV:{(**pp, y), (*p, y), (*pp, p)}

S4:--------------------
TREF: {p, *pp}
TGEN: {*p, y}
DEP:{
    p: S3--->S4
    *pp: S3--->S4
}
TDEF:{(*p, S4), (*pp, S3), (p, S3), (pp, S2), (y, S4)}
TEQUIV:{(**pp, y), (*p, y), (*pp, p)}
```

```
 S5:--------------------
 TREF: {p, *p, *pp, pp, y}
 TGEN: {**pp, y}
 DEP:{
     p: S3--->S5
     *p: S4--->S5
     *pp: S3--->S5
     pp: S2--->S5
     y: S4--->S5
     y: S4-O->S5
 }
 TDEF:{(**pp, S5), (*p, S4), (*pp, S3), (p, S3), (pp, S2), (y, S5)}
 TEQUIV:{(**pp, y), (*p, y), (*pp, p)}
```

icpp3

```
109062320@hopper:~/hw2$ make run
`/home/share/llvm_build/bin/llvm-config --bindir`/opt -disable-output -load-pass-plugin=./hw2.so -passes=hw2 icpp3.ll
S1:--------------------
TREF: {}
TGEN: {p}
DEP:{}
TDEF:{(p, S1)}
TEQUIV:{(*p, x)}

S2:--------------------
TREF: {}
TGEN: {pp}
DEP:{}
TDEF:{(p, S1), (pp, S2)}
TEQUIV:{(**pp, x), (*p, x), (*pp, p)}

S3:--------------------
TREF: {pp}
TGEN: {*pp, p}
DEP:{
    pp: S2--->S3
    p: S1-0->S3
}
TDEF:{(*pp, S3), (p, S3), (pp, S2)}
TEQUIV:{(**pp, y), (*p, y), (*pp, p)}

S4:--------------------
TREF: {p, *pp}
TGEN: {*p, y}
DEP:{
    p: S3--->S4
    *pp: S3--->S4
}
TDEF:{(*p, S4), (*pp, S3), (p, S3), (pp, S2), (y, S4)}
TEQUIV:{(**pp, y), (*p, y), (*pp, p)}
```

```
S5:--------------------
TREF: {*pp, pp, p}
TGEN: {**pp, y}
DEP:{
    *pp: S3--->S5
    pp: S2--->S5
    p: S3--->S5
    y: S4-0->S5
}
TDEF:{(**pp, S5), (*p, S4), (*pp, S3), (p, S3), (pp, S2), (y, S5)}
TEQUIV:{(**pp, y), (*p, y), (*pp, p)}
```

foo

```
109062320@hopper:~/hw2$ make run
`/home/share/llvm_build/bin/llvm-config --bindir`/opt -disable-output -load-pass-plugin=./hw2.so -passes=hw2 foo.ll
S1:--------------------
TREF: {c, b}
TGEN: {a}
DEP:{}
TDEF:{(a, S1)}
TEQUIV:{}

S2:--------------------
TREF: {}
TGEN: {p}
DEP:{}
TDEF:{(a, S1), (p, S2)}
TEQUIV:{(*p, y)}

S3:--------------------
TREF: {c, b}
TGEN: {d}
DEP:{}
TDEF:{(a, S1), (d, S3), (p, S2)}
TEQUIV:{(*p, y)}

S4:--------------------
TREF: {y, x, d, a, *p}
TGEN: {f}
DEP:{
    d: S3--->S4
    a: S1--->S4
}
TDEF:{(a, S1), (d, S3), (f, S4), (p, S2)}
TEQUIV:{(*p, y)}

S5:--------------------
TREF: {y, x, d, a, *p}
TGEN: {g}
DEP:{
    d: S3--->S5
    a: S1--->S5
}
TDEF:{(a, S1), (d, S3), (f, S4), (g, S5), (p, S2)}
TEQUIV:{(*p, y)}
```

```
S6:--------------------
TREF: {i, p}
TGEN: {*p, y}
DEP:{
    p: S2--->S6
}
TDEF:{(*p, S6), (a, S1), (d, S3), (f, S4), (g, S5), (p, S2), (y, S6)}
TEQUIV:{(*p, y)}

S7:--------------------
TREF: {y, x, d, a, *p}
TGEN: {h}
DEP:{
    y: S6--->S7
    d: S3--->S7
    a: S1--->S7
    *p: S6--->S7
}
TDEF:{(*p, S6), (a, S1), (d, S3), (f, S4), (g, S5), (h, S7), (p, S2), (y, S6)}
TEQUIV:{(*p, y)}

S8:--------------------
TREF: {y, *p}
TGEN: {f}
DEP:{
    y: S6--->S8
    *p: S6--->S8
    f: S4-0->S8
}
TDEF:{(*p, S6), (a, S1), (d, S3), (f, S8), (g, S5), (h, S7), (p, S2), (y, S6)}
TEQUIV:{(*p, y)}
```

Experiment report – how you implement the pass

TREF

When iterating through all the instructions in a Basic Block, if an instruction is a store instruction, I use a function called `handling_TREF` to identify all elements in TREF(Si), where Si represents the ith statement. Additionally, I have divided the

function into several parts:

1. For the Right Hand Side of the assignment:
    - All variables used in `BinaryOperator` instructions.
    - Pointers with dereference operators.
    - Values obtained after dereferencing.
2. For the Left Hand Side of the assignment:
    - Pointers with dereference operators.
3. Updating the set with elements in TEQUIV.

For the data structure that stores the data in the set, I use a vector called `all_TREF` to store the TREF set for each statement. Additionally, I have a struct named `TREF`, which contains a vector to store the elements in the set.

TGEN

For efficiency, when processing the Left Hand Side (LHS) in the `handling_TGEN` function, I make sure to include the relevant variable in the set.

After the `handling_TREF` function completes its process, I then invoke `handling_TGEN`. This subsequent step updates the set with elements identified in TEQUIV, ensuring a comprehensive reflection of all relevant variable interactions and equivalences.

Regarding the data structure for storing data in the set, it is similar to the one used for TREF part. Specifically, the data is stored in a vector named `all_TGEN`.

Dependences

After executing `handling_TGEN`, I proceed with `handling_dependency` to manage data dependencies. This function utilizes a struct named `DEP` to store the name of the variable (`VAR_name`) and the source (`src`) and destination (`dst`) of the dependency.

In the context of flow dependency, the function iterates over the elements in `all_TREF` for the current statement (indexed by `stmt_cnt - 1`). It checks for intersections between the elements in `all_TREF` and `TDEF`. If an intersection is found, it implies a flow dependency, record the variable name, its source (derived from `TDEF`), and its destination.

Similarly, for output dependencies, the function iterates through the elements in `all_TGEN` for the same statement. It then checks for intersections between the elements in `all_TGEN` and `TDEF`.

Finally, these vectors, `flow_dep` and `out_dep`, representing the flow and output dependencies, respectively, are added to global vectors `all_flow_dep` and `all_out_dep`.

TDEF

In `handling_TDEF`, called right after `handling_dependency`, I update `TDEF` using `all_TGEN[stmt_cnt - 1]`. `TDEF` is a map pairing each variable with the statement number of its latest assignment, like (p, S1) for variable 'p' assigned in statement 1. For each element in `all_TGEN`, if it's already in `TDEF`, I update its associated statement number. If it's new, I add it to `TDEF`.

TEQIV

In the `handling_TREF` function, after dealing with TREF and TGEN elements, I manage TEQIV for pointers. When both the store location and the value to be stored are pointers, and the dereference level is appropriate, I either update an existing entry in `TEQIV` or create a new one. This entry includes the dereference level, names of the elements, and their relationship.

Then, the `handling_alias` function is called. This function checks and updates aliases in `TEQIV`. If conditions are met (e.g., pointer levels and existing entries), it adds new aliases to `TEQIV`, reflecting the indirect relationships between variables.

Finally, `update_equiv` is used to propagate changes in equivalences throughout `TEQIV`. If an entry has aliases, it updates the corresponding elements in other entries to maintain consistency across the equivalences.