Summary of Pinpoint and IFDS/IDE

The document summarizes the problem domains of Pinpoint and IFDS/IDE framework, and demonstrate the feasibility of adapting IFDS/IDE problems to Pinpoint. Advanced bugs, which are out of the problem domain of Pinpoint, are briefly introduced with an example of loop hoisting.

Meanwhile, the layers of pointer analysis in the IFDS/IDE framework and Pinpoint are compared to point out the difference in terms of the way to use the result of pointer analysis and their precision. Although these two styles of approaches are both based on layer design, IFDS/IDE framework does not suffer *pointer trap*, because top clients of data flow analysis are path insensitive and do not rely on sparse analysis, which makes Pinpoint distinguished from it.

The conclusion is delivered in the end. Pinpoint reduces the value flow analysis to the constrained reachability problem in the symbolic expression graph, and processes the attributes of general graph representation and uniform bug specification, which support flexible scheduling of a collection of bug detectors. Based on the observation, an IFDS/IDE problem can be formulated as a value flow problem in the problem domain of Pinpoint.

Part 1: Problem Domain of IFDS/IDE

The solving of IFDS/IDE problem is formulated by two components

- A set of data flow fact
  - IFDS: A finite set of dataflow fact `D`
  - IDE: Environment `Env` mapping dataflow elements to values in a lattice.
- Flow function
  - IFDS: Gen-Kill function
  - IDE: Environment transformer

Example

IFDS Problem

According to whether the form of the flow function depends on the input or not, IFDS problem can be classified into two types. For separable problems, the function in the program can be summarized by a bit-vector function `f(x) = (x - Kill) \cup Gen`, where `Kill` and `Gen` are constant [7].

- Separable problems:
  - Reaching definition
  - Available expression
  - Live variables
- Non-separable problems

    - Possibly-uninitilized variables
    - May/Must-alias analysis
    - Truly-live variables
    - Copy-constant propagation
    - Taint analysis

IDE Problem

- Constant propagation [10]
- Software product line analysis [9]

Part 2: Problem Domain of Pinpoint

The detection of bug type `BType` can be defined by a set of value-flow properties `S_p = {(src; sink; psc; agg)}` [1, 2].

Modeled by Single Value-flow Property

- Bug Type 1: Null-Dereference-Like Bugs
    - (SinkMustNotReach) The operation never occurs in any path, i.e., `agg=never`
    - Example
        - Null pointer dereference
        - Bad buffer size From System function
        - Divide by zero
        - Free of non-heap memory
        - Use of uninitialized variables
        - Integer overflow
        - File descriptor use after free (Taint-style)
        - Data race (Taint-style)
    - Remark: DivideByZero and NPD are similar

Example The value-flow property of NPD is

`null-deref:=(v = malloc(_); _ = *v,*v = _;v = 0; never)`

- Bug Type 2: Memory-Leak-Like Bugs
    - (SinkMustReach) The operations must occur in any path, i.e., `agg=always`
    - Example
        - Memory leak
        - File descriptor leak
- Bug Type 3: Double-Free-Like Bugs
    - Two operations never occur simultaneously in one path, i.e., `agg=non-sim`
    - Example
        - Double-free

Modeled by Compositional Value-flow Property

- Buffer overflow (SinkMustNotReach)
    - Buffer copy without checking input size
    - Incorrect calculation buffer size

Conclusion

- The value-flow property is defined by a 4-tuple `(src; sink; psc; agg)`. Bug specification is a composition of one or more value-flow properties.
- The bugs with the same value-flow properties distinguished from each other in terms of the type of value and the operations on them in the program, although the conditions of triggering the bugs might be totally the same, e.g., DividedByZero and Null Pointer Dereference.

Part 3: Model IFDS/IDE Problem in Pinpoint Style

IFDS problem

- Copy-constant propagation: Given a program `P` and a variable `x` at a certain point `lc`, determine whether the variable `x` has the same value at `lc` for any program execution. The constant value does not depend on the execution.

    ```
    //Syntactic assumption: a = const; a = b;
    int a = 0;
    int b = 1;
    if (c) {
      a = b;
    }
    //whether a is constant or not
    ```

    - IFDS/IDE framework collects all constant integers to form `D` and obtain the facts reachable at `lc`.
    - For Pinpoint, two SEG traces: `0 -> a` and `1 -> b -> a` The analysis demands the comparison of two traces. It should be noticed that searching strategy affects the problem domain of Pinpoint. Specifically, the comparison can be reduced to searching SEG paths in two directions, and the path `1 -> b -> a -> 0` indicates that the property is violated.

IDE problem

- Constant propagation: Generalize the program to any arithmetic program [10].
    - IDE-style framework records the computation in the environment `Env`
    - It is straightforward to adapt Pinpoint by collecting constant-value in the constraints, which records the mapping of variables in `Env` in essence.

Conclusion

- Searching stragety affects the domains of problems in Pinpoint. In different configurations, the searching direction and constraint types determine the border of problem domains.

Part 4: Advanced Bugs

For other types of bugs, such as loop hoisting [5], it is out of the problem domain of Pinpoint, because the specification of these bugs can not be formulated by value-flow properties.

```
int incr(int x) {
  return x + 1;
}

// incr will not be hoisted since it is cheap(constant time)
void foo_linear(int size) {
  int x = 10;
  for (int i = 0; i < size; i++) {
    incr(x); // constant call, don't hoist
  }
}

// call to foo_linear will be hoisted since it is expensive(linear in size).
void symbolic_expensive_hoist(int size) {
  for (int i = 0; i < size; i++) {
    foo_linear(size); // hoist
  }
}
```

Part 5: Pointer-Analysis Layer

Pinpoint

Pinpoint outperforms IFDS/IDE framework in terms of path sensitivity. In the phase of SEG construction, path conditions are encoded in a sparse manner, i.e., irrelevant statements are cut out according to the data dependency, so that the scalable analysis can be achieved with high precision.

The calculation of data dependency relies on the results of pointer analysis. Imprecise pointer analysis causes spurious data dependency, and degrades the precision improvement of sparse analysis. In order to solve this pointer trap, Pinpoint offloads path sensitive analysis to the detection of specific bug, i.e.,

- Step 1: Perform intraprocedural flow-sensitive pointer analysis and interprocedural parameter-return points-to summary.

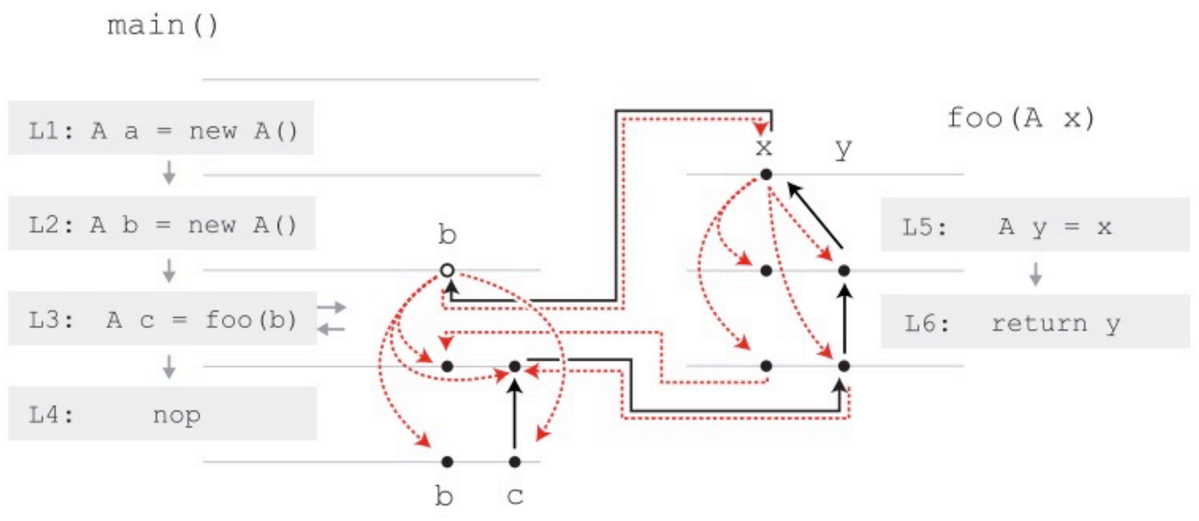
- Step 2: Obtain data dependance based on points-to relation and construct Symbolic Expression Graph(SEG)
- Step 3: Traverse SEG and collect constraints.

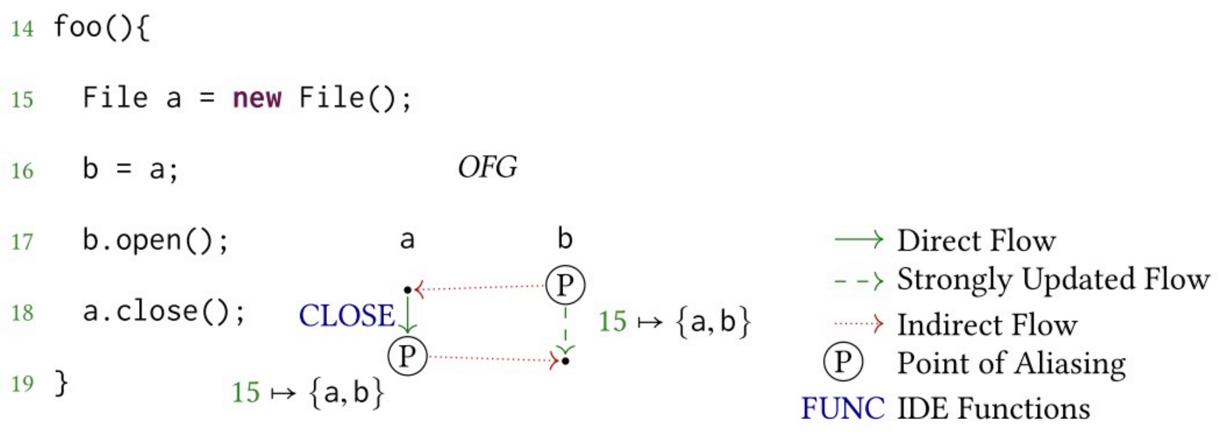
IFDS/IDE Framework

Path insensitivity makes IFDS/IDE framework escape from pointer trap. For a particular analysis, such as typestate analysis, the preprocess of pointer analysis is required to aliasing.

One of the typical work is *IDEal* [9] and it takes advantages another IFDS/IDE framework *Boomerang* for pointer analysis [4]. The analysis takes the following steps:

- (Boomerang) Backward analysis to identify the allocation sites.
- (Boomerang) Forward analyisis to obtain aliasing relation.
- (IDEal) During the phase of value-flow propagation, aliasing relation is utilized to propagate in-direct value flow.





```
foo(){
  File a = new File();
  b = a;
  b.open();
  a.close();
}
```

The examples illustrate the procedure of computing aliasing relation and value-flow propagation.

Conclusion

- The layers of pointer analysis in these two frameworks both perform flow-sensitive analysis. The differences include
  - The usage of result: Pinpoint utilizes it to construct SEG for sparse analysis while

  IFDS/IDE framework takes advantage of it to propagate value flow fact in a full manner.
  - The demand of precision: Pinpoint calls for path sensitive result and offload the overhead to clients to mitigate pointer trap. IFDS/IDE framework focus on flow- and context-sensitivity rather than path sensitivity.

Conclusion

- Pinpoint reduces the value flow analysis to the constrained reachability problem in the symbolic expression graph.
  - Given a symbolic expression graph(SEG), the detection of bug type `BType` can be defined by a set of value-flow properties `S_p = {(src; sink; psc; agg)}`
  - Constraint 1(Reachability Constraint)

    The environment `CondEnv` computes the conditions of the path in SEG, from `src` to `sink`. `src` and `sink` are reachable if and only if there is a path such that the condition in `CondEnv` implies `psc`.
  - Constraint 2(Aggregation Constraint)
    - `agg=never`: There is no reachable path from any one pair of source and sink.
    - `agg=always`: The disjunction of condition from any one of source to all the sinks is valid.
  - The paths violating these two constraints are the evidence of the presence of bug `BType`
- Attribute of Pinpoint
  - General program representation, i.e., SEG is dependent to bug type.
  - Uniform bug specifications, i.e., bug specifications are formulated by value-flow properties, based on which bugs can be grouped according to the sinks and pre-conditions.
- Advantage over IFDS/IDE framework
  - Precision: Pinpoint benefits from the sparse analysis and achieves path sensitive with low cost, while exploded super graph does not store the branch condition and yield path insensitive result.
  - Efficiency: Uniform bug specification supports the multiple bug detection in one pass and searching process of reachable paths can even be optimized according to mutual synergies.
- Relationship of problem domain
  - The problem domain of Pinpoint subsume IFDS and IDE problems.

Appendix and Memo: Optimization in Pinpoint

For multi-bug detection, it is costly to perform the analysis for each bug. Pinpoint support the detection of all bug types in one check. The observation is that the bugs with the same sinks can be detected simultaneously, in which the searching process of one type of bug can be terminated soundly according to the information collected when searching SEG traces for the other type.

The similar observation is the inconsistent pre-condition provides extra information to reduce the overhand of constraint solving.

- Dual Group 1(Same sink)
  - Memory leak
  - Free global pointer
- Dual Group 2 (Inconsistent pre-condition)
  - Memory leak
  - Null pointer dereference

Reference

1. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code (PLDI 2018)
2. Conquering the Extensional Scalability Problem for Value-Flow Analysis Frameworks (ICSE, 2020)
3. IDEal: efficient and precise alias-aware dataflow analysis (OOPSLA 2017)
4. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java (ECOOP 2016)
5. Infer: List of all issue types (link)
6. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis.
7. Precise interprocedural dataflow analysis via graph reachability
8. Program analysis via graph reachability
9. IDEal : efficient and precise alias-aware dataflow analysis (OOPSLA 2017)
10. SPLLIFT- Statically analyzing software product lines in minutes instead of years (PLDI 2013)
11. Precise interprocedural dataflow analysis with applications to constant propagation