

On the K -reachability Problem for Data Flow Analysis

Reasoning may/must queries for combinatorial definitions

Xiao Xiao, Charles Zhang

Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

November 29, 2012

What We Do in this Study?

We classify the data flow problems and locate our position.

Classification

Data flow problems can be classified into two categories under the Gen/Kill framework:

- Separable Problems, for which, the *Kill* information is statically determined. All the bit-vector problems, such as reaching definition, liveness analysis, are in this category;
- Non-separable Problems, in contrast, the *Kill* information cannot be determined in prior. An example is points-to analysis, in which, a statement $q = p$ killing q points-to o is decided by whether or not p points to o , which is unknown before points-to analysis.

Classification

Data flow problems can also be classified by the way of tracking the properties of the variables:

- Independent Attribute Analysis: We obtain the data flow facts for different variables independently. For example, for x and y , we obtain $[x = a \text{ or } x = c]$ and $[y = b \text{ or } y = d]$.
- Relational Attribute Analysis: We keep track of the dependent properties between different variables. For example, we obtain $[x = a \text{ and } y = b]$ or $[x = c \text{ and } y = d]$.

For simplicity, we use the term *1-simultaneity* to indicate the independent attribute analysis. The term *K-simultaneity* where $K > 1$ to represent the relational attribute analysis.

Classification

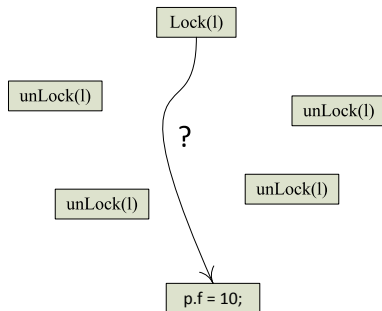
Muth *et.al* have done a comprehensive study for the independent/relational non-separable analysis [3]. From their paper, precise non-separable analysis is very hard (PSPACE-hard + EXPTIME-hard). In this paper, we study the *relational separable problems*, as the independent separable problems are the main theme of the bit-vector algorithms. We can always use our solutions to approximate the non-separable problems by performing a value flow analysis for both normal and pointer variables first. Of course, this incurs loss of precision.

Motivating Examples

Relational separable problem is a building block for many important problems.

Lockset Analysis

Which locks must/may be held at the statement $p.f = 10$?



Must hold: Every path from $lock(l)$ to $p.f = 10$ excludes $unlock(l)$.

May hold: There is a path from $lock(l)$ to $p.f = 10$ (reaching definition).

Nested Lock Analysis

Can the locks l_1 and l_2 be simultaneously held at $p.f = 10$?

```
lock(  $l_1$  );  
if (  $a > b$  ) {  
    unlock(  $l_1$  );  
    lock(  $l_2$  );  
}  
p.f = 10;
```

Locks l_1 and l_2 are *separately* held at $p.f = 10$, but they can not be *simultaneously* held.

This nested locking information is essential for deadlock analysis.

Memory Leak Analysis

Example

```
int *p = malloc(); //  $o_1$ 
if ( p == NULL ) return -1;
int *q = malloc(); //  $o_2$ 
if ( q == NULL ) return -1;
.....
free(p);
free(q);
return 0;
```

To reveal the leakage of o_1 , we should figure out if there *exists* an execution path starting from the o_1 allocation site to the program termination that never releases o_1 . Of course, all the pointers have been resolved before the leak analysis.

Leak Analysis is basically a non-separable problem. However, we first replace the $free(p)$ statement by $free(o_1)$ with a points-to analysis, the rest of the problem is a separable problem. Double free analysis, dereferenced freed pointer analysis all bear the similarity.

Conditional Store Analysis

The definitions of $p1$ and $q2$ do not reach $p3.f = q3$ simultaneously.

Example

```
UDT p1 = new UDT();
UDT q1 = new UDT();
if ( a > b ) {
    UDT t1 = p1;
    p2 = q1;
    q2 = t1;
}
p3 =  $\phi$ (p1, p2);
q3 =  $\phi$ (q1, q2);
p3.f = q3;
```

Path sensitively analyzing $p3.f = q3$ yields two assignments:

```
p1.f = q1;
p2.f = q2;
```

The following statements are spurious in flow sensitive analysis:

```
p1.f = q2;
p2.f = q1;
```

This analysis can be used to improve the precision of pointer analysis and value flow graph. We will review this example later.

Boolean Verification

We try to conclude that q is always non-null when the dereference $q.sum$ is executed in the following code:

Example

```
if ( p != null ) {  
    q = p;  
}  
.....  
if ( p != null ) {  
    q.sum += 10;  
}
```

Boolean Abstraction:

$p_b = \text{true} \Leftrightarrow p == \text{null}$

$q_b = \text{true} \Leftrightarrow q == \text{null}$

```
if ( p_b == false ) {
```

```
    q_b = p_b;
```

```
}
```

```
.....
```

```
if ( p_b == false ) {  
    assert( q_b == false );  
}
```

Boolean Verification

However, after the abstraction, only the abstract domain becomes smaller, the problem is still a non-separable problem. Next, we use an information flow analysis and transform the program again:

Example

```
if (  $p_b == \text{false}$  ) {  
     $q_b = p_b$ ;  
}
```

.....

```
if (  $p_b == \text{false}$  ) {  
    assert(  $q_b == \text{false}$  );  
}
```

```
if (  $p_b == \text{false}$  ) {  
     $q_b = \text{false}$ ;  
}
```

.....

```
if (  $p_b == \text{false}$  ) {  
    assert(  $q_b == \text{false}$  );  
}
```

We decide if the state ($p_b == \text{false}$, $q_b == \text{true}$) reaches the assert.

Summary

All the sample problems are instances of the K -simultaneity problem, which can be addressed by K -reachability analysis with different requirements:

Problem	Quantifier	Resolving Predicates	Simultaneity
Lockset Analysis	\exists, \forall	NO	1, 2
Nested Lock Analysis	\exists, \forall	NO	> 1
Memory * Analysis	\exists, \forall	NO	1, 2
Cond. Store Analysis	\exists	NO	> 1
Boolean Verification	\exists	YES	≥ 1

Why we study it again, haven't it been solved yet?

Previously, only heavy algorithms (high time and programming complexity) are known for the K -reachability problem ¹. We try to theoretically characterize the K -reachability problem and figure out simple solutions if possible.

Methodology	Handle Non-distributive Data Flow Facts	Variable Correlation	Complexity	Demand Driven
Bit-vector Framework	NO	NO	P	NO
Path Sensitive Value Flow Analysis e.g. BEBOP [1]	YES	YES	PSPACE-hard $O(E \times T^3)$	NO
Regular Path Queries [2]	NO	NO	$O(G \times P \times sts \times pars)$	YES
Ours	NO	YES	P NP-hard Co-NP-hard	YES

¹Actually, our formal definition of K -reachability is more general, which cannot be solved by any of the algorithms here directly.

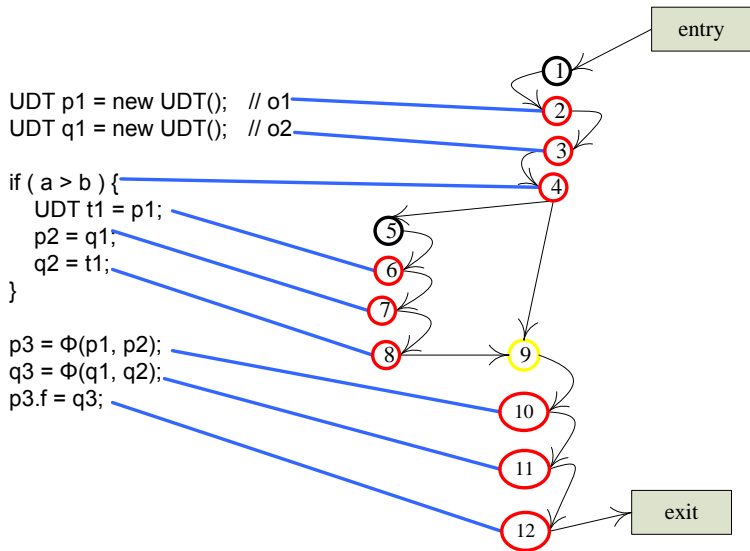
Formalization of Our Work

Preliminary: Program Points

Definition (Augmented Control Flow Graph)

The Augmented Control Flow Graph (ACFG) explicitly expresses the program points in the control flow graph. Every statement has two program points, the in-point $inP(s)$ and the out-point $outP(s)$. For simplicity, the two consecutive statements $s1$ and $s2$ share one program point: $outP(s1) = inP(s2)$. Meanwhile, we put every statement s into its $outP(s)$ point. This is because every statement only affects those program points after the execution of that statement.

Example



Preliminary

Definition (SSA Representation)

SSA IR numbers the definitions to the same variable x at different places to x_1, x_2, \dots . Moreover, SSA allows at most one definition of each variable to be visible at any program point. To ensure this property, at proper places, the SSA IR puts a ϕ node to merge the different definitions of the same variable and produces a new definition for that variable. We call the ϕ definition virtual definition and the programmer written definition concrete definition.

Preliminary

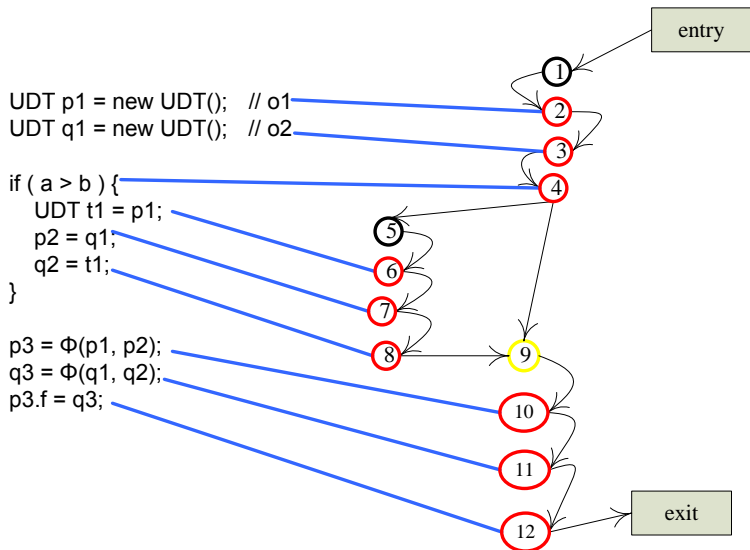
Definition (SSA Representation)

SSA IR numbers the definitions to the same variable x at different places to x_1, x_2, \dots . Moreover, SSA allows at most one definition of each variable to be visible at any program point. To ensure this property, at proper places, the SSA IR puts a ϕ node to merge the different definitions of the same variable and produces a new definition for that variable. We call the ϕ definition virtual definition and the programmer written definition concrete definition.

But, there is one exception. The in-point for the ϕ statement may see multiple definitions of the same variable. To simplify our discussion, we assume that this kind of program points are not queried.

See the example again...

Note the yellow node



Reaching Definition

Definition (Existential)

Give a definition $d: x = \text{expr}$ and a program point B , d is a existential reaching definition at B iff there is a execution path reaching B through d and d is the last write to x in the path.

Existential reaching definition is also termed as d is visible at B .

Definition (Universal)

Give a definition $d: x = \text{expr}$ and a program point B , d is a universal reaching definition at B iff for all execution paths reaching B through d , d is the last write to x .

Reaching Definition: Classical Solution

Classical gen/kill framework for reaching definition problems:

$Q = (\mathbb{D}, \mathbb{L}, \mathbb{F}, \mathbb{I}, \sqcap)$:

\mathbb{D} : a set of definition statements

\mathbb{L} : a lattice of $2^{\mathbb{D}} \cup \{\top, \perp\}$

\mathbb{F} : a set of transfer functions

\mathbb{I} : the initialization of data flow facts for every program point

\sqcap : the meet operator.

For existential problem, \sqcap is the *set union operator*. For universal problem, \sqcap is the *set intersection operator*.

We use $rd(B)$ to represent the set of reaching definitions at B .

We use $rpp(d)$ to represent the set of program points that the definition d can reach.

Simultaneity

In the classical formulation, any two definitions d_1 and d_2 in $rd(B)$ are *separately* computed, *i.e.* d_1 and d_2 may take different paths to B . As shown in our motivating examples, we also need to answer whether or not d_1 and d_2 can reach B *simultaneously* via the same path. Therefore, we need a more general formulation.

Modeling Simultaneity: K -reachability

Definition (Existential)

The existential K -reachability query $E_k(\mathbb{D}_k, B, \rho)$, where:

$\mathbb{D}_k : \{a_1, a_2, \dots, a_k\}$,

B : a program point,

$\rho : \mathbb{D}_k \rightarrow \mathbb{D}_k \cup B$,

determines if there is a path to B passes all the definitions in \mathbb{D}_k . And, for any definition $a_i \in \mathbb{D}_k$, a_i must be visible at $\rho(a_i)$.

Definition (Universal)

The universal K -reachability query $U_k(\mathbb{D}_k, B, \rho)$ determines if all the paths to B allow the definitions $a_i \in \mathbb{D}_k$ visible at corresponding $\rho(a_i)$.

We refer to the special case, where $\forall a_i, \rho(a_i) = B$, i.e. all the definitions simultaneously reach B . We denote the special cases as E_k and U_k .

K -reachability: Our Goal

We first try to characterize the K -reachability problem in the complexity hierarchy. For the P problems, we try to design efficient algorithms for them. These algorithms should have better scalability than existing solutions.

Resolving Predicates?

Unfortunately, being aware of the predicate expressions is hard. So we do not consider it in the rest of the study.

Considering predicates is hard

Resolving path predicates considers the impact of a conditional statement to the afterward execution, which is highly demanded by verification tools. The traditional approximation for path feasibility is that *every branch can be taken regardless of the prior statements in the path.*

Example

```
if ( p != null ) {  
    q = p;  
}  
.....  
if ( p != null ) {  
    q.sum += 10;  
}
```

Boolean Abstraction:

$p_b = \text{true} \Leftrightarrow p == \text{null}$

$q_b = \text{true} \Leftrightarrow q == \text{null}$

```
if ( p_b == false ) {
```

$q_b = p_b;$

```
}
```

.....

```
if ( p_b == false ) {
```

$\text{assert}(q_b == \text{false});$

```
}
```

Considering predicates is hard

Unfortunately, if we consider the predicates, the existential K -reachability problem is NP-hard and the universal problem is co-NP-hard.

Proof Sketch: Reduction from the 3-SAT problem and the tautology problem respectively.

Considering predicates is hard

Unfortunately, if we consider the predicates, the existential K -reachability problem is NP-hard and the universal problem is co-NP-hard.

Proof Sketch: Reduction from the 3-SAT problem and the tautology problem respectively.

The NP-hardness and co-NP-hardness impede us to efficiently support the verification tools and path sensitive analysis with high precision.

Therefore, currently, we do not consider the impact from the predicates. We will go back here to design a sound approximation algorithm in this situation, in order to significantly scale the path sensitive analysis.

Scenario 1:

Do not distinguish virtual and concrete definitions

The SSA unique definition per variable property is guaranteed. Thus, both the existential and universal queries are easy.

Reaching Definition in SSA

For a statement that uses a variable v , we already know its unique reaching definition during the renaming process of SSA construction. However, how to efficiently compute the reaching definition of variable x at any out program point?

Definition

We denote B_1 dominates B_2 by $dom(B_1, B_2)$. This relation can be extracted from the dominator tree $DT(G)$ of the ACFG.

Theorem

The unique reaching definition of variable x at program point B , say x_i , is the nearest dominator of B that defines x , denoted by $ndom(x_i, B)$. x_i can be found by climbing up the $DT(G)$ from B .

Universal and Existential are equal

Lemma (Reaching Definition Enclosure)

$\forall B_1, B_2 \in rpp(a_i)$, the last definition for variable a on any of the $B_1 \rightsquigarrow B_2$ path, if exists, would be a_i .

This lemma states that **any** path between two points of $rpp(a_i)$ preserves the definition a_i . Therefore, $E_k = U_k$ in this scenario. We refer to U_k only in the rest of this section.

K -reachability

To answer U_k , where $\forall a_i, \rho(a_i) = B$, we have two questions:

- In which order the definitions a_1, a_2, \dots, a_k are appeared in the path?
- How to quickly determine there is a path between two program points B_1 and B_2 that preserve a given set of definitions?

Ordering Problem

Lemma (Unique Order)

We permute a_1, a_2, \dots, a_k as follows:

1. $\forall a_i, a_j$, either $\text{dom}(a_i, a_j)$ or $\text{dom}(a_j, a_i)$.
2. Ordering a_1, a_2, \dots, a_k by placing a_i before a_j if $\text{dom}(a_i, a_j)$.

$E_k(a_1, a_2, \dots, a_k, B, \rho)$ and $U_k(a_1, a_2, \dots, a_k, B, \rho)$ returns true iff there exists a path passing a_1, a_2, \dots, a_k in the order Ω .

This lemma tells us there is only one way to go through all the definitions.

Universal K -reachability

Theorem (Chaining Algorithm)

$$U_k(a_1, a_2, \dots, a_k, B, \rho) \Leftrightarrow \\ \text{ndom}(\Omega_1, \Omega_2) \wedge \text{ndom}(\Omega_1, B) \wedge \\ \text{ndom}(\Omega_2, \Omega_3) \wedge \text{ndom}(\Omega_2, B) \wedge \\ \dots \wedge \\ \text{ndom}(\Omega_k, B)$$

Where $\forall a_i, \rho(a_i) = B$. Ω_i is the i^{th} definition obtained by the unique order lemma.

The chaining algorithm has to evaluate only $2K - 1$ ndom relations, thereof the complexity is $O(K)$. It can also be generalized to answer the universal queries without the restriction $\forall a_i, \rho(a_i) = B$.

Scenario 2:

Virtual Definitions are NOT Counted as Definitions

The SSA property unique definition per variable is violated. Therefore, querying is much harder in this case.

Universal Query

However, universal query is still simple because, a program point seeing many definitions violates the *universal* requirement.

Universal Reachability Lemma

Lemma (UR Lemma)

$$U_1(a, B) \Leftrightarrow \text{ndom}(a, B).$$

Proof.

\Rightarrow

a is on all the paths from Entry to B .

$\therefore \text{dom}(a, B)$.

$\therefore \forall a \rightsquigarrow B$, there is no other definitions to a .

$\therefore \text{ndom}(a, B)$.

\Leftarrow

apparent. □

Therefore, if we require a concrete definition a_1 arriving B and B is dominated by a virtual definition a_2 , then a_1 cannot universally reach B .

Universal K -reachability

Therefore, all the conclusions presented in Scenario 1 still work for the universal queries here. That's to say,

Universal K -reachability

Therefore, all the conclusions presented in Scenario 1 still work for the universal queries here. That's to say,

The universal query is amazingly easy!

Existential Query (Not Completely Solved, $\nabla_{_}\nabla$)

It is hard in this case because we cannot determine in which order the K definitions appear in the path. However, verifying the feasibility of an order is in P.

Existential K -reachability: Complexity

Even all branches are considered feasible, the situation for existential query is not better. I still cannot prove or disprove the NP-hardness of the E_k problem, but a piece of bad news is that its weaker form is NP-hard.

Existential K -reachability: Complexity

Even all branches are considered feasible, the situation for existential query is not better. I still cannot prove or disprove the NP-hardness of the E_k problem, but a piece of bad news is that its weaker form is NP-hard.

Definition

The existential selection K -reachability problem $ES_k(\mathbb{P}, \mathbb{S}, B)$ is:

\mathbb{P} : the partition of the statements by their defined variables, e.g.

$\{x_1, x_2, \dots\}$ is the set of the statements defining variable x . We assume it has m partitions and we refer to the i^{th} partition as \mathbb{P}_i .

\mathbb{S} : it contains m sets and $\mathbb{S}_i \subseteq \mathbb{P}_i$. There are k sets in \mathbb{S} are non-empty.

The problem is to determine the existential of a path to B that $\forall i$, at least one definition in \mathbb{S}_i is visible at B .

Existential K -reachability: Complexity

Example

```
UDT p1 = new UDT();
UDT q1 = new UDT();
if ( a > b ) {
  UDT t1 = p1;
  p2 = q1;
  q2 = t1;
}
.....
p3 =  $\phi$ (p1, p2);
q3 =  $\phi$ (q1, q2);
p3.f = q3;
.....
```

The partition for variable p is $\{p1, p2\}$, for q is $\{q1, q2\}$. Before we enumerate which combination of p and q ($\{p1, q1\}$, $\{p1, q2\}$, $\{p2, q1\}$, $\{p2, q2\}$) will reach $p3.f = q3$, we first answer the question: if any of the combinations is feasible? This is essentially the ES_k query.

Existential K -reachability: Complexity

Theorem

The ES_k problem is NP-hard.

Proof Sketch: Reduction from the Hamilton path problem. Every node in the graph is mapped to a variable. For every incoming edge $x \rightarrow y$ of y , we built a statement that defines y . Of course, it invalidates other definitions to y .

Since the E_k query is the extreme case of the $ES_k(\mathbb{P}, \mathbb{S}, B)$ problem where each set in \mathbb{S} has at most one element, thereof I guess the E_k problem is hard.

Existential K -reachability: Good News

E_k is hard because we cannot determine an order of the definitions in \mathbb{D} that appear in the path. If we specify an order, verifying the feasibility of the order is in P.

Lemma

A path $x \rightsquigarrow y$ never kill a definition a on the way iff $\forall z \in x \rightsquigarrow y, z \in rpp(a)$.

Definition (def-Carrying Search)

The task is to search a $x \rightsquigarrow y$ path where all the definitions a_1, a_2, \dots, a_k are preserved. We start by putting x into a worklist. An edge $u \rightarrow v$ is valid iff $\forall a_i, a_i \notin rd(v)$. Then, we put v to the worklist and stop until y is reached.

K-reachability: Good News

For a query $E_k(\mathbb{D}_k, B, \rho)$, we search the path between a_i, a_{i+1} with the def-Carrying search. This process can be parallelized since the choice of the path $a_i \rightsquigarrow a_{i+1}$ does not impact validity of the path $a_{i+1} \rightsquigarrow a_{i+2}$.

The search efficiency can be enhanced by a A* or bidirectional BFS search, especially for the case source and target are far on the CFG. The complexity is of course $O(kn)$. Since it is easily parallelizable, we use k threads, the overall time is $O(n)$. We are currently working on designing an efficient index structure to speedup the search.

Applications

Pointer Analysis

SSA lends flow sensitivity to the flow insensitive points-to analysis. With K -reachability analysis, we can lend path sensitivity to a flow insensitive points-to analysis, by preprocessing the constraints before submitting them to the points-to solver. We assume the target language is Java because of simplicity. C programs are also applicable.

Pointer Analysis

SSA lends flow sensitivity to the flow insensitive points-to analysis. With K -reachability analysis, we can lend path sensitivity to a flow insensitive points-to analysis, by preprocessing the constraints before submitting them to the points-to solver. We assume the target language is Java because of simplicity. C programs are also applicable.

Example

```
UDT p1 = new UDT();
UDT q1 = new UDT();
if ( a > b ) {
    UDT t1 = p1;
    p2 = q1;
    q2 = t1;
}
p3 =  $\phi$ (p1, p2);
q3 =  $\phi$ (q1, q2);
p3.f = q3;
```

Pointer Analysis

SSA lends flow sensitivity to the flow insensitive points-to analysis. With K -reachability analysis, we can lend path sensitivity to a flow insensitive points-to analysis, by preprocessing the constraints before submitting them to the points-to solver. We assume the target language is Java because of simplicity. C programs are also applicable.

Example

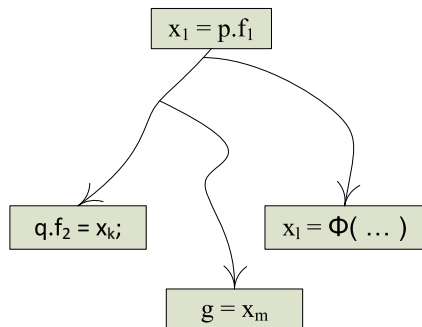
```
UDT p1 = new UDT();
UDT q1 = new UDT();
if ( a > b ) {
    UDT t1 = p1;
    p2 = q1;
    q2 = t1;
}
p3 =  $\phi$ (p1, p2);
q3 =  $\phi$ (q1, q2);
p3.f = q3;
```

Pointer Related Statements:

```
p3 =  $\phi$ (p1, p2);
q3 =  $\phi$ (q1, q2);
x1 = q3.f;
x2 = x1;
x3 = x2;
p3.f = x3;
```

Pointer Analysis

Suppose the program has been converted to SSA form. We consider a chain of assignments starting with a load statement or ϕ statement. The end of the chain is a global, a store or a ϕ statement. We can extract a tree shape widget by merging the chains with the same start statement.



In the sample tree widget, we show how to process the chain with store statement as end point.

Pointer Analysis

Consider the assignment chain:

$q.f_2 = x_k = x_{k-1} = x_{k-2} = \dots = x_1 = p.f_1$. The assignments are all established iff the $E_k(\mathbb{D}, B, \rho)$ is true, where $\mathbb{D}_k = \{p, x_1, x_2, \dots, x_k, q\}$, $B : q.f_2 = x_k, \rho(p) = x_1, \rho(x_1) = x_2, \dots, \rho(x_k) = q.f_2, \rho(q) = q.f_2$.

Fortunately, the definitions are partially ordered: a path should pass p, x_1, \dots, x_k in order. The only exception is the definition of q , we do not know its position. Therefore, for different combinations of the definitions of p and q , e.g. p_2 and q_1 , we try q_1 with all possible positions in the sequence p, x_1, \dots, x_k . The complexity is not high because inserting q_1 between x_i and x_{i+1} only do def-Carrying search for $x_i \rightsquigarrow q_1$ and $q_1 \rightsquigarrow x_{i+1}$.

Pointer Analysis

After the co-reachable pairs of p and q are identified, we re-construct the constraints:

1. We merge the variables x_1, \dots, x_k , because they are equivalent variables. Suppose the merged variable is named X .
2. For every definition p_i , we create a constraint $X_i = p_i.f_1$.
3. For every co-reachable pair p_i and q_j , we create $q_j.f_2 = X_i$ if X_i exists.

After the transformation, we directly send the constraints to a flow insensitive points-to solver. This approach can also be used to construct a SSA like partial path sensitive program representation.

Pointer Analysis

The K -reachability way of thinking path sensitivity captures the major precision impacts by the path sensitive analysis, and meanwhile, it avoids the exponential grows of the paths in traditional BDD based enumeration analysis. Moreover, our preprocess does not interpret the load and store statements, hence it can be integrated into either whole program algorithms or modular algorithms, making them know nothing about paths but gaining path benefits.

Lockset Analysis

We can obtain the must locksets via the U_1 query. We first modify the ACFG:

- We first replace every function call with its must lock summary;
- We rewrite the lock statement $lock(l)$ to $l = 1$ and the $unlock(l)$ to $l = 0$.
- If there are must alias lock statements, for example, $lock(l_1)$ and $lock(l_2)$ must operate on the same lock, we add a new node X in ACFG and put a statement $l = 1$ in it, and, remove $l_1 = 1$ and $l_2 = 1$. Then, for every flow edge flowing into $lock(l_1)$ or $lock(l_2)$, we redirect it to X . Finally, we connect X to both $lock(l_1)$ and $lock(l_2)$.
- We do the same thing for unlock must aliases.

After all, we perform a SSA transformation on the program and perform the U_1 query.

Nested Lock Analysis

We do the same transformation as in the lock analysis. The next thing is of course performing the E_2 and U_2 queries to obtain the may and must nested locksets.

Thank you!

This research is a brand new project launched recently, we only have preliminary results now. Your comments are highly appreciated.



T. Ball and S. K. Rajamani.

Bebop: a path-sensitive interprocedural dataflow engine.

In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 97–103, New York, NY, USA, 2001. ACM.



Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu.

Parametric regular path queries.

In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 219–230, New York, NY, USA, 2004. ACM.



R. Muth and S. Debray.

On the complexity of flow-sensitive dataflow analyses.

In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 67–80, New York, NY, USA, 2000. ACM.