

On the Importance of Program Representations in Static Analysis

Xiao Xiao

Supervisor: Charles Zhang

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

February 25, 2013

Basic Concepts

Definition (Static Analysis)

Static analysis is a set of automatic or semi-automatic techniques to gather the semantic knowledge of a program without executing it.

Definition (Program Representation)

The encoded form of static analysis results is called program representation. Usually, we either build additional graphical data structures to encode the results, or, we directly inject the semantic information into the code and construct an intermediate representation (IR).

Our survey consists of four parts:

- 1 Terminologies.
- 2 Motivating examples and introduction to surveyed contents.
- 3 Graphical data structures.
 - Loop nesting forest.
 - DJ-graph.
 - Program structure tree.
 - Program dependence graph.
- 4 SSA Intermediate representations and its variants: HSSA, SSI

Terminology

Basic Concepts

Assuming the following concepts you are familiar with:

- Control flow graph (CFG);
- DFS decomposition of a graph: tree edge, forward edge, backward edge, cross edge;
- Strong connected component (SCC);
- Dominance, immediate dominance, strict dominance, post-dominance, and dominate tree;

Dominance Frontier

Definition

The dominance frontier $DF(x)$ of node x is the set of all nodes y such that x dominates a predecessor of y but does not strictly dominate y . Moreover, $DF(S) = \bigcup_{x \in S} DF(x)$.

Definition (Iterated dominance frontier (IDF))

IDF(S) for a set S of nodes is the limit of the following sequence:

$$\begin{aligned} IDF(S) &= DF(S) \\ IDF(S) &= DF(S \cup IDF(S)) \end{aligned}$$

Loop

- 1 A **loop** is a strongly connected component (SCC).
- 2 A loop is a **natural loop** iff for every backward edge $x \rightarrow y$ in any DFS decomposition, $y \text{ dom } x$.
- 3 There is a *unique* node x in a loop that dominates all other nodes in the same loop. x is the **loop header**.
- 4 z is an **entry node** of a loop if there is an edge $y \rightarrow z$, where y is not the loop member.
- 5 An **irreducible loop** has multiple entry nodes. The loop header is a unique entry node of a reducible loop.
- 6 A program is **reducible** if all loops are natural loops.
- 7 Two loops in a reducible program are either nested (all nodes in one loop appear in another loop) or disjoint (no node in common).

Control Dependence

A node w is said to be **control dependent** on edge $u \rightarrow v$ if:

- 1 w post-dominates v ;
- 2 if $w \neq u$, then w does not post-dominate u .

Intuitively, control dependence means that if the execution flows along $u \rightarrow v$, the control will eventually reach w .

Motivating Examples

Induction Variable Elimination + Loop Code Motion

```
int a[SIZE], b[SIZE];
```

```
void f (void)
```

```
{
```

```
  int i1, i2, i3;
```

```
  for (i1 = 0, i2 = 0, i3 = 0; i1 < SIZE; i1++) {
```

```
    a[i2++] = input();
```

```
    for (i3 = SIZE; i3 > i2; --i3)
```

```
      a[i3] = a[i2] * a[i2];
```

```
  }
```

```
}
```

```
int a[SIZE], b[SIZE];
```

```
void f (void)
```

```
{
```

```
  int i1, i3, temp;
```

```
  for (i1 = 0, i3 = 0; i1 < SIZE; i1++) {
```

```
    a[i1] = input();
```

```
    temp = a[i1] * a[i1];
```

```
    for (i3 = SIZE; i3 > i2; --i3)
```

```
      a[i3] = temp;
```

```
  }
```

Challenges:

- 1 Which are induction variables?
- 2 Which are equivalent induction variables?
- 3 Which variables or expressions are loop invariant?
- 4 What is the loop scope of an invariant variable or expression?

Graphical Representations

Our former example requires exhaustive analysis of loops. In fact, all of the following control flow information is important:

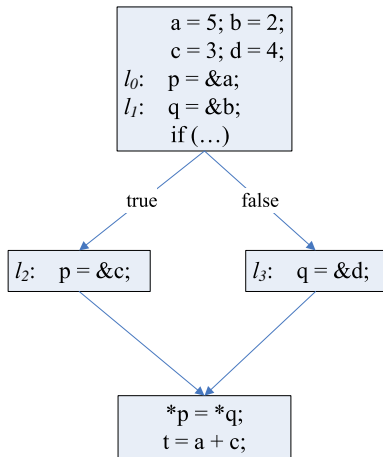
- 1 Loop structure \implies Loop nesting forest.
- 2 Dominance relations \implies DJ-graph.
- 3 Self-similarity substructures \implies Program Structure Tree.
- 4 Control dependence relations \implies Program dependence graph.

In this survey, we will study four data structures that can help us analyze the information above.

Information Flow Analysis

What is the value of t ?

Is t a constant?



Challenges:

- 1 Does the statement `*p = *q` impact the value of t ?
- 2 Which variables p and q may point to *respectively*?
- 3 Which variables p and q may point to *simultaneously*?

Intermediate Representation

The former example requires the analysis of how the variables are defined and used. Other than obtained the define and use information, we also requires the analysis algorithm being *efficient*, *scalable*, and *precise*.

The single static assignment(SSA) form meets our goal by:

- 1 Efficient and scalable \implies Sparse analysis.
- 2 Precise \implies Separating the definitions of the same variable.

Important SSA variants are HSSA and SSI. HSSA handles pointer induced side effects and SSI can transparently incorporate predicate information into analysis.

Loop Nesting Forest

What is loop nesting forest?

The **loop nesting forest**, informally, is a tree representation for the loop containment relations.

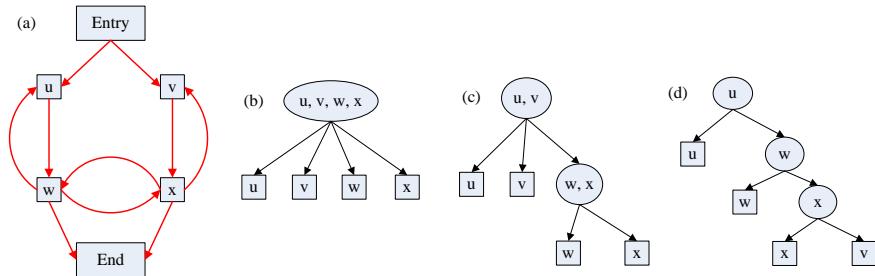


Figure : Illustration of different loop nesting forests. The loop headers are shown in the ellipse, none cycle members of the loop are shown in the rectangle. (a). A sample CFG. (b). Screedhar-Gao-Lee forest. (c). Steensgard's forest. (d). Havlak's forest.

Constructing loop nesting forest

We have different types of loop nesting forest because, in an irreducible program, loops may *overlap* but not always be nested or disjoint. There is no consensus of how to interpret the the overlap relationship. Therefore, different guidelines produce different forests.

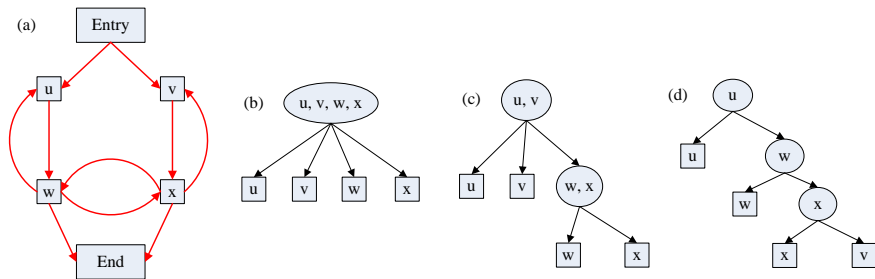
However, we can give a framework to construct any of the forest shown previously. The parameter of the framework is *the way to pick the loop headers*.

Constructing loop nesting forest

Suppose we know how to pick the headers, the loop nesting forest \mathcal{L} is generated from the CFG G_0 as follows:

1. Identifying all the maximal SCCs in G_0 as the outermost loops. Consider any SCC X , an edge from a vertex inside X to one of its headers is referred to as a **loopback edge** of X .
2. We delete all the loopback edges from G_0 and obtain G_1 .
3. If G_1 is not cycle free, we search the SCCs in G_1 and delete the loopback edges again, until we reach a acyclic graph G_i , with $i \geq 0$.

Constructing loop nesting forest



Sreedhar-Gao-Lee's forest: picks all the vertices that are not dominated by other vertices in the same SCC as the headers;

Steensgard's forest: picks all the entry vertices to be the headers;

Havlak's forest: picks only one header from an SCC, which is the vertex having the least pre-order timestamp in the DFS traversal.

Why we need loop nesting forest?

1. It tells us the scope of a loop, which is outer loop? Which is inner loop?
2. It tells us how the data flow is processed by (going in and out) a loop;
3. Certain arbitrary problem instances (including those on irreducible graphs) can be transformed into equivalent problem instances based on acyclic graphs.

Application: Building Dominator Tree

The dominator tree for a directed acyclic graph (DAG) is easy to compute due to the following theorem:

Lemma

On a DAG, the immediate dominator of node w is the least common ancestor of w 's predecessors on the dominator tree.

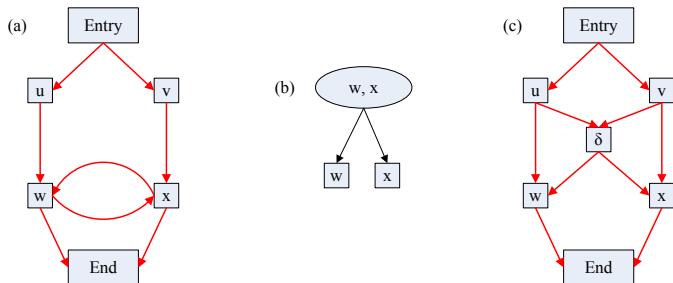
Therefore, we can process the vertices in topological order and maintain a dynamic structure for querying the least common ancestor on tree.

Application: Building Dominator Tree

The acyclic algorithm can be easily generalized to reducible cyclic graphs. The key is *the loopback edges in a reducible graph do not impact the dominance relations for any nodes*. Therefore, we remove the loopback edges and apply the acyclic algorithm, then we obtain a dominator tree for reducible graph.

Application: Building Dominator Tree

Generalizing the acyclic algorithm to irreducible graphs requires loop nesting forest, and, suppose we use Steensgard's. For every loop top down in the forest, we transform it as follows:



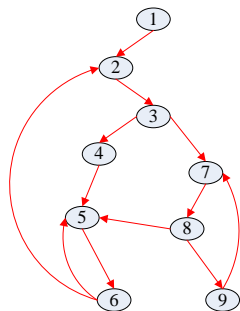
The transformed graph is acyclic. We apply our acyclic algorithm to compute the dominator tree T . The dominator tree for original graph is simply deleting all the artificial nodes (δ node in our example) from T .

DJ-Graph

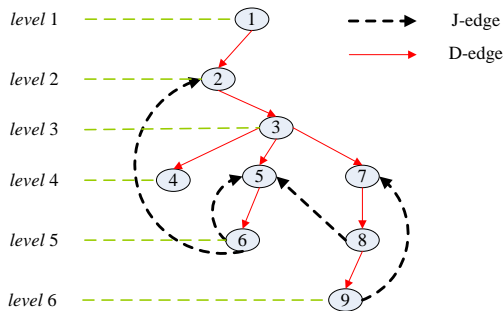
What is DJ-graph?

Definition (DJ-graph)

A **DJ graph** is just the dominator tree of a CFG augmented with join edges. A CFG edge is a join edge (J-edge) if it has no corresponding edge on the dominator tree. A dominator tree edge is called D-edge.



(a). a control flow graph



(b). DJ graph

Why we need to know DJ-graph?

1. DJ-graph is a very powerful tool to compute any dominance related properties, such as dominance frontier, iterative dominance frontier, multiple-node dominance relation, merge sets, *etc.*
2. It gives a fast way to do incremental data flow analysis. Because, from DJ-graph, we can obtain an optimal re-iteration order in elimination based data flow analysis.

Dominance Frontier

Definition

The dominance frontier $DF(x)$ of node x is the set of all nodes y such that x dominates a predecessor of y but does not strictly dominate y .

Lemma

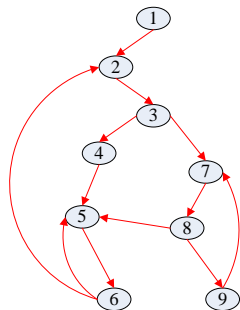
To compute $DF(x)$, we simply visit all the vertices y in the subtree of x . For every J -edge $y \rightarrow z$, $z \in DF(x)$ iff $z.level \leq x.level$.

Theorem

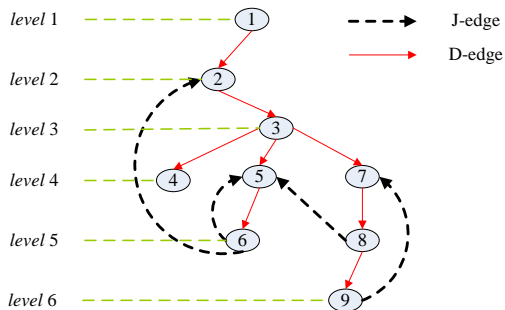
$$DF(x) = \bigcup_{y \in subtree(x)} \{w \mid w \in DF(y) \wedge w.level \leq x.level\} \\ \cup \{t \mid x \xrightarrow{J} t \wedge t.level \leq x.level\}$$

Example

We compute $DF(7)$. There are two J-edges in the subtree of 7: $9 \rightarrow 7$ and $8 \rightarrow 5$. Both 7 and 5 have the same level label with 7, hence, $DF(7) = \{5, 7\}$.



(a). a control flow graph



(b). DJ graph

Iterated Dominance Frontier

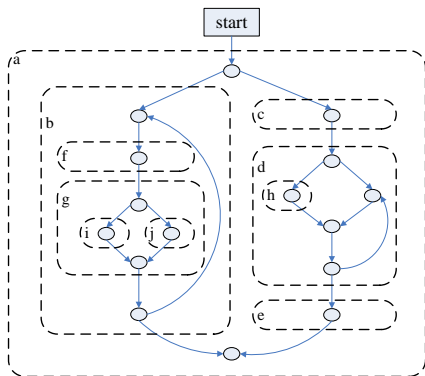
- 1 To compute $IDF(S)$, we first compute $DF(x)$, $\forall x \in S$ and then, $IDF(S) = \bigcup_{x \in S} DF(x)$. The complexity is $O(n^2)$ for this method.
- 2 The inefficiency is that, *a node z may appear in $DF(x_1), DF(x_2), \dots$, merging these sets will visit z multiple times.*
- 3 With DJ-graph, we can avoid the revisit of z multiple times. Therefore, the overtime becomes $O(n)$.

Program Structure Tree

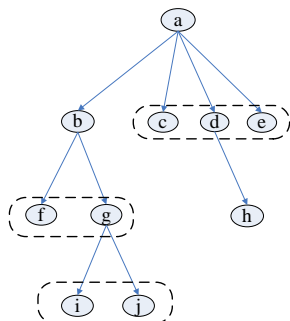
What is PST?

Definition

Program Structure Tree (PST) is a hierarchical decomposition of CFG. Each node in PST is a single entry single exit (SESE) region, which is a self-similarity structure to CFG.



a. control flow graph with SESE regions



b. program structure tree

Why PST is useful?

- 1 *Global structure*: Each SESE region is a control flow graph in its own right. Therefore, any procedural analysis algorithm can be applied unchanged to each SESE region.
- 2 *Local structure*: PST localizes the effect of irreducible structure to several SESE regions. For reducible regions, we can apply reducible algorithms to speed up analysis.
- 3 *Sparsity*: If the summarized transfer function for a SESE region is only an identity function w.r.t a data flow analysis, such transparent region can be directly bypassed, resulting in quick propagation of data flow facts.

Program Dependence Graph

What is PDG?

Definition

*Program dependence graph (PDG) is a combination of the **control dependence graph (CDG)** and the **data dependence graph (DDG)**. The DDG subgraph is exactly the graphical representation of the def-use chains. The CDG subgraph answers the question that under what conditions, a statement will be executed.*

PDG is almost the standard structure used by every software engineering tools in the pre-SSA age. However, since its DDG subgraph is too large (SSA is a factored DDG) and incomplete for pointer languages, only the CDG subgraph is popular now.

Program Sums

1. read(n);
 2. i = 1;
 3. sum = 0;
 4. while (i <= n) do
 5. j = 1;
 6. while (j <= i) do
 7. sum = sum + j;
 8. j = j + 1;
 9. end
 10. i = i + 1;
 11. end
 12. write(sum);
- End Sums

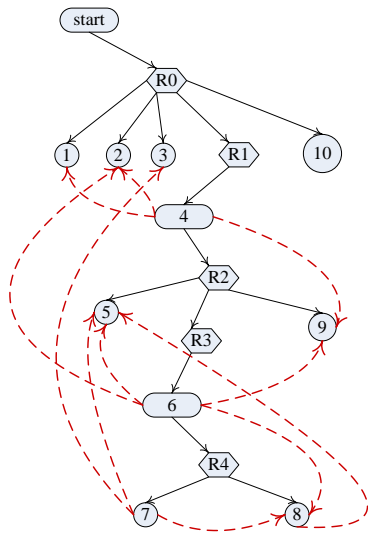


Figure : Rounded rectangle represents a control statement (e.g. if statement), the circle is a non-control statement, and the hexagon is a region node.

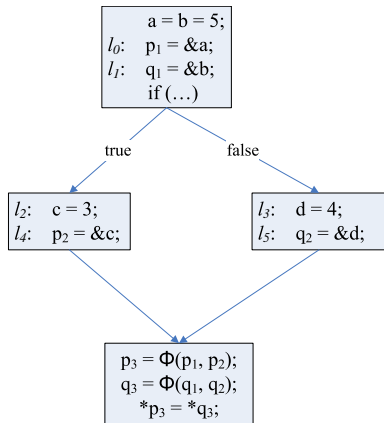
Why we need PDG?

PDG is another way to hierarchically represent the program:

- 1 The control dependence relation is the key to summarize the code behavior. Therefore, PDG is widely used in code clone detection and plagiarism detection, because cloned code often does not change the control structure.
- 2 Forward and backward slicing is as simple as a reachability analysis on PDG.
- 3 PDG is still the top cited data structure in fault localization. Because, we always identify a control region to be a faulty place.

Single Static Assignment Form

Constructing SSA



Procedure:

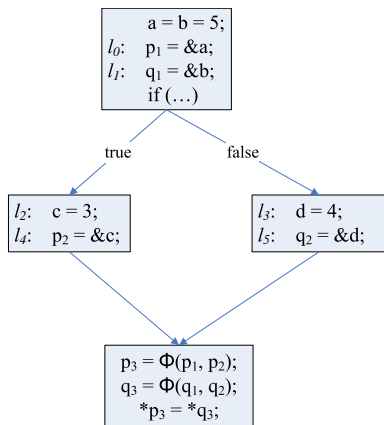
- 1 Renaming variables to ensure that every variable is defined only once.
- 2 Using ϕ function to merge the multiple reaching definitions for the same variable. Now every use of a variable sees a unique definition.
- 3 Eliminating dead ϕ functions to produce **pruned SSA**.

Why we should use SSA?

- 1 SSA enables sparse program analysis. Sparse analysis is a data flow information minimization strategy.
- 2 SSA version sparse analysis automatically translates a flow insensitive analysis to partial flow sensitive analysis, which retains high precision for analysis results.
- 3 Register allocation is simpler and produces better code in SSA form.

Sparse Analysis and Precision Degredation

Explanations:



- The points-to information of p_1 at the points l_3 and l_5 is the same as l_0 . Therefore, we only maintain points-to information for p_1 at l_0 . Since l_0 is the birth point of p_1 , we associate information to p_1 but not (p_1, l_0) , in sparse analysis.
- p_3 points to a and c , q_3 points to b and d , hence, $*p_3 = *q_3$ becomes four assignments: $a = b$, $a = d$, $c = b$, $c = d$. Obviously, $a = b$ and $c = d$ are fake.

Register Allocation

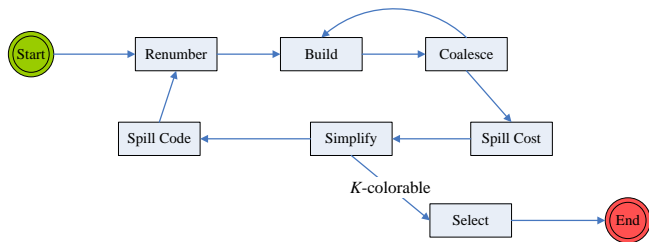


Figure : Chaintin's graph coloring register allocator.

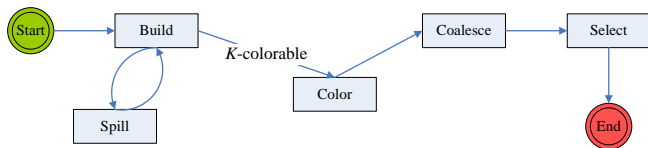


Figure : SSA based Chaintin's register allocator.

Register Allocation

Benefits of SSA register allocator:

- 1 The process is greatly simplified, because the interference graph for SSA program is *chordal*.
- 2 It always gives better allocation quality than original program. Because graph coloring is an NPC problem, the heuristics work for general problem cannot be better than the exact algorithm for chordal graph.

SSA Extensions: HSSA and SSI

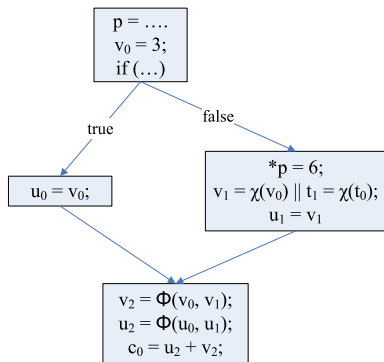
Why we need HSSA?

Definition

Hashed SSA Hashed SSA (HSSA) adds two functions χ and μ to annotate the pointer and function call induced may-def and may-use side effects. The translated code has no implicit data modifications.

For pointer languages, raw SSA format is inadequate for full sparse analysis. Being sparse is extremely important for interprocedural analysis, such as memory leak detection and unsafe code detection. HSSA has broad applications in such defect analysis scenario.

Constant Propagation

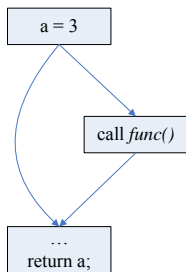


Explanations:

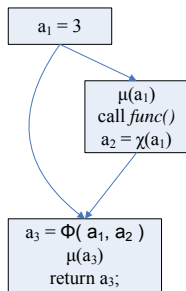
- 1 The statement $v_1 = \chi(v_0) \parallel t_1 = \chi(t_0)$ indicates that $*p = 6$ may change the variable v_0 and t_0 .
- 2 Now we know u and v is not constant before executing $c = u + v$, in original program;
- 3 HSSA suffers from the same merge before use problem as SSA: c is a constant but we cannot infer this fact.

Zero Version

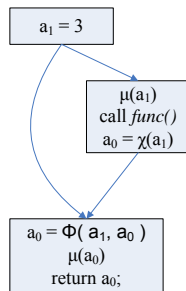
Inserting the χ and μ functions into the program will blow up the program size. However, some pseudo variables do not impact the final data flow results. Therefore, we merge these variables to maximize the benefits for analysis algorithms and minimize the program size. The merged variables are called *zero versions*.



a. Original program



b. SSA form



c. SSA form with zero versions

What is SSI?

Not only the variable definition is the information source, variable use can also be:

- 1 Backward data flow analysis such as liveness analysis collects and propagates variable use information. We cannot perform sparse analysis on SSA for backward data flow problems.
- 2 The predicate expression such as *if* ($p > 0$) is also useful for path sensitive defect analysis. However, SSA does not support the integration of predicate information into sparse analysis.

Constructing SSI

Building SSI form involves adding pseudo assignments for a variable v :

- 1 ϕ : at control flow merges, exactly the SSA ϕ -function;
- 2 σ : at locations where control flow splits and at least one of the disjoint paths from the split uses the value of v .

The control flow split points are obtained through the *iterated post-dominate frontiers*, which is a parallel concept to *iterated dominance frontier*. The *pruned SSI* is always constructed in practice because a program may have 6 times σ functions more than the ϕ functions.

Backward Sparse Analysis: Very Busy Analysis

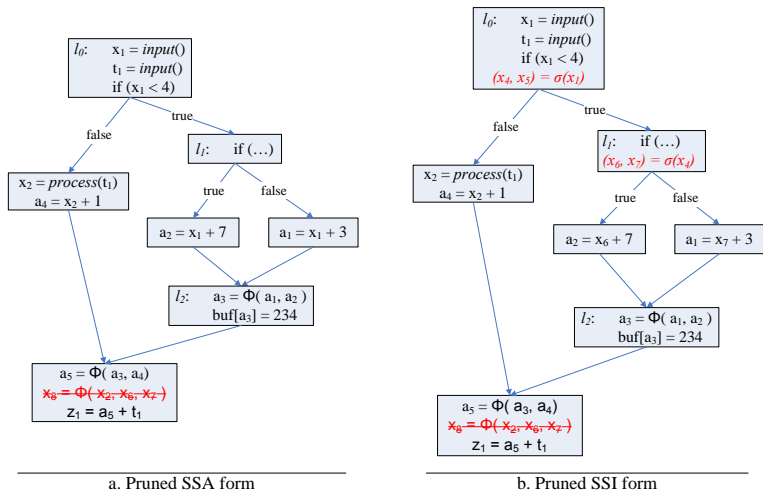


Figure : Is x_1 very busy at l_1 ? Sparse analysis on SSA only obtains the very busy information at l_0 , but from which, we cannot infer the very busy information at l_1 .

Array Bounds Check: Requires Predicates

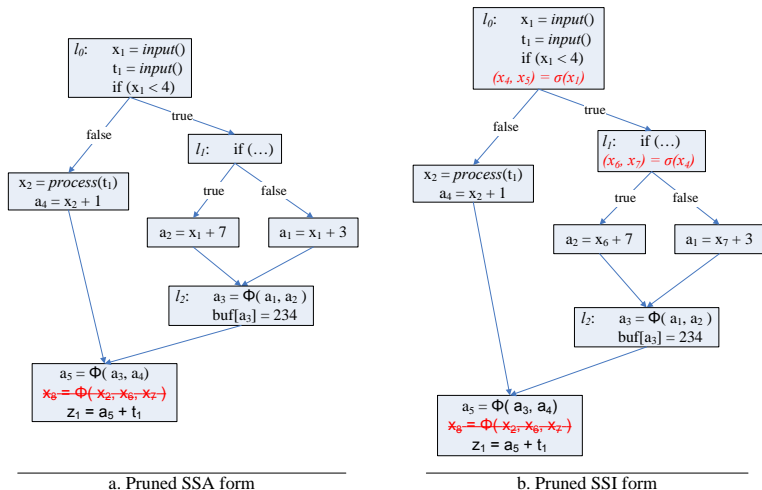


Figure : Because $x_1 < 4$, hence $a_3 < 11$. Then we can decide if $\text{buf}[a_3]$ is out of boundary.

Takeaway Ideas

Graphical Data Structures

Our surveyed data structures can help us:

- 1 If the hardest case for your problem is about loops, consider thinking *hierarchically* with the **loop nesting forest**.
- 2 If your problem is concerned with dominance relations, please try **DJ-graph**.
- 3 The **program structure tree** can help you think in a *divide-and-conquer* way, which gives parallelization opportunity.
- 4 You can compare source code semantic difference with **program dependence graph**.

SSA and Its Variants

SSA and its variants provide us:

- 1 Forward and backward sparse analysis. Sparse analysis skips irrelevant code, requires less memory for information store, and it is designed in flow insensitive style. Therefore, *sparse analysis is fast, scalable, and easy to work with.*
- 2 Reaching definitions and dominance information can be retrieved easily in **SSA**. In addition, liveness and control dependence information can also be easily retrieved in **SSI**.
- 3 Data structures based SSA might be simpler. For example, the interference graph is a *chordal* graph in SSA, it is even an *interval* graph in SSI.

Complementary Materials

Outline

This complementary slides try to answer two questions:

- 1 The research history of the surveyed data structures.
- 2 The state-of-the-art research in program representations.

Program Representations Research History

Loop nesting forest (LNF)

- In 1973, Tarjan first proposes linear time algorithm to testing the irreducibility of loops [26];
- In 1977, Hecht develops node splitting technique to handle irreducible loops in data flow analysis [13];
- In 1993, Steensgaard develops the Steensgaard LNF for code generation [25]. However, the generation algorithm is quadratic time;
- In 1996, Sreedhar *et.al* develop the Sreedhar-Gao-Lee LNF, which can be generated in linear time with DJ-graph [23];

Loop nesting forest (LNF)

- In 1997, Havalk develops the Havalk LNF, which can be constructed in almost linear time. Different to Sreedhar *et.al*'s LNF, Havalk's LNF does not requires the dominator tree in prior [12];
- In 2000, Ramalingam proposes a framework to model all the LNFs proposed by Steensgaard, Sreedhar *et.al*, and Havalk so far [19]. Based Ramalingam's previous finding, all kinds of LNF can be generated in almost linear time [18];
- In 2007, Wei *et.al* propose a algorithm that works faster in practice to construct the LNFs modeled by Ramalingam's framework [29].

DJ-graph

- In 1995, Sreedhar *et.al* first propose the DJ-graph for linear time SSA construction [22];
- More applications of DJ-graph to computing dominance relations are given in Sreedhar *et.al*'s manuscript [21];
- In 1996, Sreedhar *et.al* use DJ-graph for incremental data flow analysis [24];
- In 2002, DJ-graph is used for optimal node splitting to improve the instruction parallelism [27];
- In 2012, DJ-graph is used for fast liveness queries [8].

Program Structure Tree (PST)

- In 1994, Johnson *et.al* first propose the single entry single exit (SESE) concept and the program structure tree (PST) [14]. The construction algorithm for PST is linear, which is optimal.
- In 1997, Pingali *et.al* use PST to compute control dependence graph in linear time [17];
- In 2003, Chen *et.al* use PST to compress code [15];
- In 2011, Domínguez *et.al* use PST to dynamically generate GPU code [9];

Program Dependence Graph (PDG)

- In 1987, Ferrant *et.al* first propose PDG [10];
- In 1997, Pingali *et.al* propose the linear time algorithm to construct an encoded PDG that also answer control and data dependence queries in constant time [17].
- PDG is widely used in fault localization. The latest influential work in this category is the probabilistic PDG developed by Baah *et.al* [2].
- Later in 2006 and 2008, PDG is used for code clone detection by Liu *et.al* [16] and Gabel *et.al* [11].

Single Static Assignment (SSA)

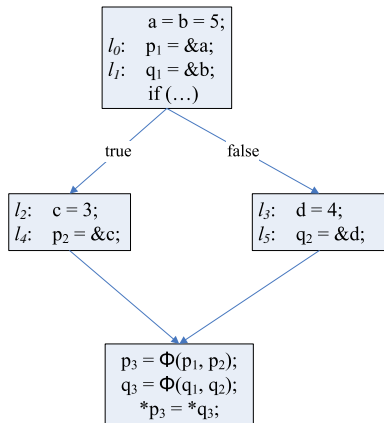
- In 1989, Cytron *et.al* first propose SSA [6];
- In 1991, based on SSA, Wegman *et.al* develop a simple and efficient constant propagation that takes conditional statements into account [28].
- In 1993, Cytron *et.al* first time consider how to control the size of SSA representation for programs with pointer induced aliases, which may square the program size [7]. However, their propose solution is specific to applications.
- In 1996, Chow *et.al* propose a general approach, HSSA representation, to handle pointer aliases and avoid the exploded program size [5].
- In 1999, Ananian *et.al* propose the static single information (SSI) form, an extension to SSA that permits backward sparse analysis [1]. However, the original SSI construction algorithm is wrong [4]. Singer *et.al* develop a correct solution [20];

State-of-the-art Research

Our research: Path-sensitive SSA (PSSA).

Problems for SSA and SSI

A sample SSA program:



Points-to analysis on SSA:

- $p_3 = q_3$ introduces fake assignments: $a = b$ and $c = d$.
- If we would like to perform sparse analysis, the fake assignments cannot be eliminated.

Therefore, sparse analysis on SSA is not very precise. Can we make it better?

Path sensitive SSA (PSSA)

Design goal:

- 1 PSSA also supports sparse analysis;
- 2 Analysis result automatically gains path sensitive precision. Just like SSA makes flow insensitive algorithm be flow sensitive.
- 3 PSSA does not significantly increase the program size.

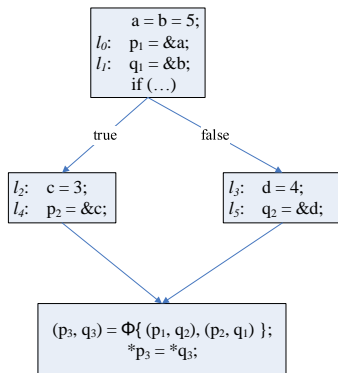
We never see any SSA extensions that implement all of these features. Bodik *et.al* [3] propose the value flow graph (VFG), which achieves path sensitive effects. However, VFG is not general enough to provide support for non-distributive data flow analysis. And, VFG is non-scalable because of its large size.

Path sensitive SSA (PSSA)

Solution proposal:

We modify the ϕ function to generate tuple but not single variable. In this way, the value correlation between p_3 and q_3 is represented explicitly.

Now, placing the ϕ functions requires a *correlated reaching definition analysis*.



Thank you!



C. Scott Ananian and Martin Rinard.

Static single information form.

Technical report, Master's thesis, Massachusetts Institute of Technology, 1999.



George K. Baah, Andy Podgurski, and Mary Jean Harrold.

The probabilistic program dependence graph and its application to fault diagnosis.

In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 189–200, New York, NY, USA, 2008. ACM.



Rastislav Bodík and Sadun Anik.

Path-sensitive value-flow analysis.

In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 237–251, New York, NY, USA, 1998. ACM.



Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello.

Ssi properties revisited.



Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich.

Effective representation of aliases and indirect memory operations in SSA form.

In Proceedings of the International Conference on Compiler Construction, volume 1060 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1996.



Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck.

Efficiently computing static single assignment form and the control dependence graph.

ACM Trans. Program. Lang. Syst., 13(4):451–490, October 1991.



Ron Cytron and Reid Gershbein.

Efficient accommodation of may-alias information in SSA form.

In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 36–45, 1993.



Dibyendu Das, B. Dupont De Dinechin, and Ramakrishna Upadrasta.
Efficient liveness computation using merge sets and dj-graphs.
ACM Trans. Archit. Code Optim., 8(4):27:1–27:18, January 2012.



Rodrigo Domínguez, Dana Schaa, and David Kaeli.
Caracal: dynamic translation of runtime environments for gpus.
In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4*, pages 5:1–5:7, New York, NY, USA, 2011. ACM.



Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren.
The program dependence graph and its use in optimization.
ACM Trans. Program. Lang. Syst., 9(3):319–349, July 1987.



Mark Gabel, Lingxiao Jiang, and Zhendong Su.
Scalable detection of semantic clones.

In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.



Paul Havlak.

Nesting of reducible and irreducible loops.

ACM Trans. Program. Lang. Syst., 19(4):557–567, July 1997.



Matthew S. Hecht.

Flow Analysis of Computer Programs.

Elsevier Science Inc., New York, NY, USA, 1977.



Richard Johnson, David Pearson, and Keshav Pingali.

The program structure tree: computing control regions in linear time.

In *Proceedings of the ACM SIGPLAN 1994 conference on*

Programming language design and implementation, PLDI '94, pages 171–185, New York, NY, USA, 1994. ACM.



Wen ke Chen, Bengu Li, and Rajiv Gupta.

Code compaction of matching single-entry multiple-exit regions.

In *In Proceedings of the 10th Annual International Static Analysis Symposium (SAS03*, pages 401–417. Springer-Verlag, 2003.



Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu.

Gplag: detection of software plagiarism by program dependence graph analysis.

In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 872–881, New York, NY, USA, 2006. ACM.



Keshav Pingali and Gianfranco Bilardi.

Optimal control dependence computation and the roman chariots problem.

ACM Trans. Program. Lang. Syst., 19(3):462–491, May 1997.



G. Ramalingam.

Identifying loops in almost linear time.

ACM Trans. Program. Lang. Syst., 21(2):175–188, March 1999.



G. Ramalingam.

On loops, dominators, and dominance frontier.

In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 233–241, New York, NY, USA, 2000. ACM.



Jeremy Singer.

Static program analysis based on virtual register renaming.
Technical report, University of Cambridge, 2006.



Vugranam C. Sreedhar, Yong fong Lee, Yong fong Leey, and Guang R. Gao.

Dj-graphs and their application to flow graph analyses, 1994.



Vugranam C. Sreedhar and Guang R. Gao.

A linear time algorithm for placing &phgr;-nodes.

In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 62–73, New York, NY, USA, 1995. ACM.



Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee.

Identifying loops using dj graphs.

ACM Trans. Program. Lang. Syst., 18(6):649–658, November 1996.



Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee.

A new framework for exhaustive and incremental data flow analysis using dj graphs.

In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation, PLDI '96*, pages 278–290, New York, NY, USA, 1996. ACM.



Bjarne Steensgaard.

Sequentializing program dependence graphs for irreducible programs. Technical report, 1993.



Robert Tarjan.

Testing flow graph reducibility.

In *Proceedings of the fifth annual ACM symposium on Theory of computing, STOC '73*, pages 96–107, New York, NY, USA, 1973. ACM.



Sebastian Unger and Frank Mueller.

Handling irreducible loops: optimized node splitting versus dj-graphs. *ACM Trans. Program. Lang. Syst.*, 24(4):299–333, July 2002.

 Mark N. Wegman and F. Kenneth Zadeck.

Constant propagation with conditional branches.

ACM Trans. Program. Lang. Syst., 13(2):181–210, April 1991.

 Tao Wei, Jian Mao, Wei Zou, and Yu Chen.

A new algorithm for identifying loops in decompilation.

In *Proceedings of the 14th international conference on Static Analysis, SAS'07*, pages 170–183, Berlin, Heidelberg, 2007. Springer-Verlag.