Static Analysis

Xiangyu Zhang

Intuition

- Instead of concretely executing the program, let's statically execute/traverse it
- Instead of producing concrete execution states, let's have abstract state related to the analysis

Analysis: Compute Data Dependence Statically

- Let's use the simplest language first

Program P ::= s
Statement s ::= s1; s2 | $x=^{L} y$ | $x =^{L} y$ op z | $x=^{L} c$ |
if (x^{L}) s1 else s2 |
while (x^{L}) s
Operation op :: = + | - | * | / | > | < | …
Value c :: = 0 | 1 | 2 … | true | false
Variable x, x1, x2, x3

Configuration

- $< s, D, X > \rightarrow < s', D', X' >$
  - Definition D: Variable -> Label
  - Dependences X: P (Label × Label)
- In contrast, dynamic dependence detection has the following configuration

$$< s, \delta, \gamma, C, D, X > \rightarrow < s', \delta', \gamma', C', D', X' >$$

  - Counter C: Label -> Int
  - Definition D: Address -> Label × Int
  - Dependences X: P (Label × Int × Label × Int)
- We call D and X in the first configuration the abstract domain, as they have nothing to do with execution
  - Note that the store is not part of the static configuration

Abstract Semantics

$$\frac{D' = D[x \mapsto L]}{< x = {}^{L}c; s, D, X > \rightarrow < s, D', X >}$$

Const-Assign

$$\frac{D' = D[x \mapsto L] \quad X' = X \cup < L, D[y] >}{< x = {}^{L}y; s, D, X > \rightarrow < s, D', X' >}$$

Copy

$$\frac{\begin{array}{c} D' = D[x \mapsto L] \\ X' = X \cup < L, D[y] > \cup < L, D[z] > \end{array}}{< x = y + z; s, D, X > \rightarrow < s, D', X' >}$$

BinOp-Add

Abstract Semantics

$$\frac{???}{< \textbf{\textit{if}}\ (x)\ \textbf{\textit{s1 else s2}}; s, D, X > \rightarrow ???}$$

If

- There is no concrete store. We do not know the branch outcomes. What can we do with the conditional statements and loop statements?
- Idea: let's explore all possible branches
  - If we have explored a branch with the same (abstract) state, we can skip the branch
  - Use a worklist

Configuration

- $< V, W, s, D, X > \rightarrow < V', W', s', D', X' >$
  - Definition D: Variable -> Label
  - Dependences X: P (Label × Label)
  - WorkList W: P (Definition × Statement)
  - Visited V: P (Definition × Statement)

Abstract Semantics

$$\frac{D' = D[x \mapsto L]}{< V, W, x = {}^{L}c; s, D, X > \rightarrow < V, W, s, D', X >}$$ Const-Assign

$$\frac{D' = D[x \mapsto L] \quad X' = X \cup < L, D[y] >}{< V, W, x = {}^{L}y; s, D, X > \rightarrow < V, W, s, D', X' >}$$ Copy

$$\frac{\begin{array}{c} D' = D[x \mapsto L] \\ X' = X \cup < L, D[y] > \cup < L, D[z] > \end{array}}{< V, W, x = y + z; s, D, X > \rightarrow < V, W, s, D', X' >}$$ BinOp-Add

Abstract Semantics

$$\frac{W' = W \cup < D, s2; s > \quad \neg < D, s2; s > \ni V \quad \neg < D, s1; s > \ni V \quad X' = X \cup < L, D[x] > \quad V' = V \cup < D, s2; s > \cup < D, s1; s >}{< V, W, if\ (x^L)\ s1\ else\ s2; s, D, X > \rightarrow < V', W', s1; s, D, X' >} \text{ If}$$

$$\frac{}{< V, W, while\ (x) s1; s, D, X > \rightarrow < V, W, if\ (x)\ s1; while\ (x)\ s1\ else\ skip; s, D, X >} \text{ While}$$

$$\frac{W = < D', s > \cup W'}{< V, W, skip, D, X > \rightarrow < V, W', s, D', X >} \text{ Next-Path}$$

An Example

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:          i=i+1
5:          sum=sum+i
      endwhile
6:  print(sum)
```

Two Importance Properties

- Does the analysis terminate
  - How does it handle loops?
- Is it sound?
  - Does it computes what it claims to compute (through a declarative statement)

Property One: Termination

- The abstraction domain is finite
  - Visited V: P (Definition × Statement)

- By definition a partial order between the abstract values, called the lattice, and showing that the computed abstract values can only change strict-monotonically, we ensure termination
  - Set subsumption

Property Two: Soundness

- Declarative statement.
  - If there is a path from L1 to L2, L1 defines variable x and L2 uses x, and there is not another definition of x along the path, L1->L2 must be in the computed dependence set.
- How to prove?
  - Our algorithm never enumerates all paths. Instead, it uses a work-list and may exercise a path from the middle

Property Two: Soundness Proof

- Trace semantics

$$< V, W, s, T, D, X > \rightarrow < V', W', s', T', D', X' >$$

  - Definition D: Variable -> Label
  - Dependences X: P (Label × Label)
  - WorkList W: P (Definition × Trace × Statement)
  - Visited V: P (Definition × Statement)
  - Trace T: Label*

Abstract Semantics

$$\frac{D' = D[x \mapsto L]}{< V, W, x = {}^{L}c; s, T, D, X > \rightarrow < V, W, s, T \cdot L, D', X >}$$

Const-Assign

$$\frac{\begin{array}{c} W' = W \cup < D, T, s2; s > \quad \neg < D, s2; s > \ni V \\ \neg < D, s1; s > \ni V \quad X' = X \cup < L, D[x] > \\ V' = V \cup < D, s2; s > \cup < D, s1; s > \end{array}}{< V, W, if\ (x^{L})\ s1\ else\ s2; s, T, D, X > \rightarrow < V', W', s1; s, T \cdot L, D, X' >}$$

If

Property Two: Soundness Proof

- Lemma: If there is a path from L1 to L2, L1 defines variable x and L2 uses x, and there is not another definition of x along the path, the evaluation must generate a trace satisfying the condition (which may not be the specific path though)
  - Lemma 1: There must be a trace leading to L1
  - Lemma 2: There must be a trace leading from L1 to L2
    - Proof by contradiction. If not, the new definition at L1 does not take the branch leading to L2
  - Lemma3: There must be a trace leading from L1 to L2 without any redefinition of x.
    - Proof by contradiction. Assume all traces from L1 to L2 have redefinition. There must be a predicate preceding the redefinition leading to L2
- Given such a trace, it must be D[x]=L1 at L2
  - Proof by contradiction

Way to Improve Our Previous Algorithm

- It is too expensive to enumerate individual paths
- How about (abstractly) execute both branches of a predicate and aggregate the results before execute the continuation?

Configuration

- $< \sout{V, W}, s, D, X > \rightarrow < \sout{V', W'}, s', D', X' >$
  - Definition D: Variable -> P(Label)
  - Dependences X: P (Label × Label)
  - ~~WorkList W: P (Definition × Statement)~~
  - ~~Visited V: P (Definition × Statement)~~

Abstract Semantics

$$\frac{D' = D[x \mapsto \{L\}]}{< x = {}^{L}c; s, D, X > \rightarrow < s, D', X >}$$

Const-Assign

$$\frac{D' = D[x \mapsto \{L\}] \quad X' = X \cup \{< L, L' > | \forall L' \ni D[y]\}}{< x = {}^{L}y; s, D, X > \rightarrow < s, D', X' >}$$

Copy

$$\frac{D' = D[x \mapsto \{L\}] \quad X' = X \cup \{< L, L' > | \forall L' \ni D[y] \cup D[z]\}}{< x = y + z; s, D, X > \rightarrow < s, D', X' >}$$

BinOp-Add

Abstract Semantics

$$\frac{X0 = X \cup \{< L, L' > \mid L' \ni D[x]\} \quad < s1, D, X0 > \rightarrow < skip, D1, X1 > \quad < s2, D, X0 > \rightarrow < skip, D2, X2 > \quad D' = D1 \cup D2 \quad X' = X1 \cup X2}{< if\ (x^L)\ s1\ else\ s2; s, D, X > \rightarrow < s, D', X' >}$$ If

$$\frac{X0 = X \cup \{< L, L' > \mid L' \ni D[x]\} \quad < s1, D, X0 > \rightarrow < skip, D1, X1 > \quad \neg D1 \subseteq D \quad D' = D \cup D1 \quad X' = X0 \cup X1}{< while\ (x) s1; s, D, X > \rightarrow < while\ (x)\ s1; s, D', X' >}$$ While

Abstract Semantics

$$\frac{\boldsymbol{X0 = X \cup \{< L, L' > \mid L' \ni D[x]\}} \quad \boldsymbol{< s1, D, X0 > \rightarrow < skip, D1, X1 >} \quad \boldsymbol{D1 \subseteq D}}{\boldsymbol{< while\ (x) s1; s, D, X > \rightarrow < s, D, X1 >}}$$

While-Exit

An Example

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:          i=i+1
5:          sum=sum+i
      endwhile
6:  print(sum)
```

Termination and Soundness

- Termination can be proved by the monotonic increase of D at while statement(s)
- Soundness
  - If there is a path from L1 to L2, L1 defines variable x and L2 uses x, and there is not another definition of x along the path, L1->L2 must be in the computed dependence set.
    - Proof sketch: for each definition d in D, we associate it with a witness trace. For each step of eval, all the definitions have their witness traces incremented by the step. We define the provenance of d the set of witness traces of d starting from the definition to its deadth (redefinitions). Then prove by contradiction that there is not a witness trace covers both L1 and L2.
- Unfortunately, not all the analyses can be optimized by evaluating both branches and then their continuation

Alias Analysis

- For each pointer variable, determine the set of global variables and the heap objects that may be pointed-to by the variable
  - One of the most important analyses

Heap Language

Program P ::= s

Statement s ::= s1; s2 | $x=^{L} y$ | $x =^{L} y$ op z | $x=^{L} c$ |
$x=^{L}$ &y | $(*x)=^{L} y$ | $x=^{L}$ *y | $x=^{L}$ malloc(y)
if (x^{L}) s1 else s2 |
while (x^{L}) s

Operation op :: = + | - | * | / | > | < | …

Value c :: = 0 | 1 | 2 … | true | false

Address a :: = 0 | 1 | 2…

Variable x, x1, x2, x3

Label L, L1, L2,…

Configuration

- $< V, W, s, T, A > \rightarrow < V', W', s', T', A' >$
  - Alias A: P (Variable × Label × P(Variable|Label))
  - PointsTo T: (Variable | Label) -> P(Variable|Label)
  - WorkList W: P (PointsTo × Statement)
  - Visited V: P (PointsTo × Statement)

```
1 x=malloc(10);
2 p=&y
3 (*x)=p
4 q=p+i
5 (*q)=x
```

Abstract Semantics

$$\frac{}{< V, W, x = {}^{L}c; s, T, A > \rightarrow < V, W, s, T, A >}\ \textbf{Const-Assign}$$

$$\frac{T' = T[x \mapsto T[y]] \quad T[y] \neq nil \qquad A' = A \cup \{< x, L, T[y] >\}}{< V, W, x = {}^{L}y; s, T, A > \rightarrow < V, W, s, T', A' >}\ \textbf{Copy}$$

$$\frac{T' = T[x \mapsto T[y]] \quad T[y] \neq nil \qquad A' = A \cup \{< x, L, T[y] >\}}{< V, W, x = y + z; s, T, A > \rightarrow < V, W, s, T', A' >}\ \textbf{BinOp-Add}$$

Abstract Semantics

$$\frac{T[y] \neq nil \quad T[z] \neq nil}{< V, W, x = y + z; s, T, A > \rightarrow < V, W, except, T, A >}$$ BinOp-Excp

$$\frac{T' = T[x \mapsto \{y\}] \quad A' = A \cup \{< x, L, y >\}}{< V, W, x = {}^{L}\&y; s, T, A > \rightarrow < V, W, s, T', A' >}$$ Addr-of

$$\frac{T' = T[x \mapsto \{L\}] \quad A' = A \cup \{< x, L, L >\}}{< V, W, x = {}^{L}\&malloc(y); s, T, A > \rightarrow < V, W, s, T', A' >}$$ Malloc

$$\frac{T' = T[\forall t \in T[x], t \mapsto T[t] \cup T[y]] \quad \forall t \in T[x] \wedge isVar(t), A' = A \cup \{< t, L, T[y] >\}}{< V, W, (* x) = {}^{L}y; s, T, A > \rightarrow < V, W, s, T', A' >}$$ Pnt-Write

Strong update versus weak update

Abstract Semantics

$$\frac{\begin{array}{c} T' = T[x \mapsto \{T[t] | \forall t \in T[y]\}] \\ A' = A \cup \{< x, L, \{T[t] | \forall t \in T[y]\} >\} \end{array}}{< V, W, x = {}^{L}(* y); s, T, A > \rightarrow < V, W, s, T', A' >}$$

Pnt-Read

Abstract Semantics

$$\frac{W' = W \cup < T, s2; s > \quad \neg < T, s2; s > \ni V \quad \neg < T, s1; s > \ni V \quad V' = V \cup < T, s2; s > \cup < T, s1; s >}{< V, W, if\ (x^L)\ s1\ else\ s2; s, T, A > \rightarrow < V', W', s1; s, T, A >} \text{ If}$$

$$\frac{}{< V, W, while\ (x) s1; s, T, A > \rightarrow < V, W, if\ (x)\ s1; while\ (x)\ s1\ else\ skip; s, T, A >} \text{ While}$$

$$\frac{W = < T', s > \cup W'}{< V, W, skip, T, A > \rightarrow < V, W', s, T', A >} \text{ Next-Path}$$

Example

```
1:    p=&x
2:    q=malloc(10)
3:    *q=p
4:    t=q+2
5:    *t=malloc(5)
6:    r=(*t)
7    (*r)=&p
```

A More Efficient Alias Analysis

- Traversing different paths is very expensive
- How about we ignore the control flow
  - Eliminate strong update
    - x =… never overwrites the PointsTo set of x, but rather add to it

x=&y
x=&z
t=x

PointsTo ->: (Variable | Label) X (Variable|Label)
Eg. x -> y (read as x points to y)

Analysis Rules

$$\frac{x = \&y}{x \to y}$$

$$\frac{x = {}^{L}malloc(y)}{x \to L}$$

$$\frac{x = y \quad y \to t}{x \to t}$$

$$\frac{(*x) = y \quad y \to t \quad x \to p}{p \to t}$$

$$\frac{x = y + z \quad y \to t}{x \to t}$$

$$\frac{x = (*y) \quad y \to t \quad t \to q}{x \to q}$$

Example

```
1:    p=malloc (10)
2:    (*p)=&x
3:    q=p+1
4:    (*q)=&y
5:    r=q+1
6:    (*r)= &z
7:    i=0;
8:    t=p;
9:    while  (i<3) {
10:       z=(*t);
11:       t=t+2;
12:       i=i+1;
13: }
14: x=(*z);
```

Flow Sensitive and Flow Insensitive Analysis

- With and without respecting control flow
- The analyses we have learned, except the preceding alias analysis, are flow sensitive
- Other flow insensitive analysis
  - Type inference

Context Sensitive and Context Insensitive Analysis

Program P ::= Global vd; fd; s
VarDef vd ::= x | vd; x
Function f :: = M(y) { s }
FuncDef fd :: = f | fd; f
FuncId M, M1, M2, …
Statement s ::= s1; s2 | $x=^{L} y$ | $x =^{L} y$ op z | $x=^{L} c$ |
if (x^{L}) s1 else s2 |
while (x^{L}) s | call (M, x) ^{L}

Operation op :: = + | - | * | / | > | < | …
Value c :: = 0 | 1 | 2 … | true | false
Variable x, x1, x2, x3

Semantics

- $< s, \delta, C, \mu >$
  - Context C: Label*
  - Stack μ: Context X Var -> Value
  - isGlobal (x): Var -> Bool

Semantics Rules

$$\frac{\mu' = \mu[< C, x >\mapsto c] \quad \neg isGlobal(x)}{< x = c; s, \delta, C, \mu >\rightarrow< s, \delta', C, \mu >}$$

Local-Const-Assign

$$\frac{\delta' = \delta\left[x \mapsto \mu[C, y]\right] \quad \neg isGlobal(y) \quad isGlobal(x)}{< x = y; s, \delta, C, \mu >\rightarrow< s, \delta', C, \mu >}$$

GL-Copy

$$\frac{M(y)\,\{s'\} \quad C' = C \cdot L \quad isGlobal(x) \quad \mu' = \mu[< C', y >\mapsto \delta[x]] \quad < s', \delta, C', \mu' >\rightarrow< skip, \delta', C', \mu'' >}{< call(M, x)^L; s, \delta, C, \mu >\rightarrow< s, \delta', C, \mu >}$$

Call

Context Sensitivity in Alias Analysis

```
A () {
1:    B(&x, &y)
2:    B(&z, &t)
}
```

```
B (p,q) {
10:   (*p)=q
}
```

Alias Analysis Rules with Functions

$$\frac{x = \&y}{x \to y} \qquad \frac{x = {}^{L}malloc(y)}{x \to L}$$

$$\frac{x = y \quad y \to t}{x \to t} \qquad \frac{(*x) = y \quad y \to t \quad x \to p}{p \to t}$$

$$\frac{x = y + z \quad y \to t}{x \to t} \qquad \frac{x = (*y) \quad y \to t \quad t \to q}{x \to q}$$

$$\frac{call(M, x) \quad M(y)\,\{s\} \quad x \to t}{y \to t}$$

Context Sensitive Alias Analysis

- The impression of context insensitive analysis comes from that information from different contexts is undesirable merged
- Context sensitive alias analysis

CPointsTo ->: Context X (Variable | Label) X (Variable|Label)

Eg. $C \vdash x \rightarrow y$ (read as x points to y in context C)

CStatement :Context X Statement

Eg. $C \vdash s$ (read there is statement s in context C)

Context Sensitive Alias Analysis

$$\frac{C \vdash x = \&y}{C \vdash x \to y} \qquad \frac{C \vdash x = {}^{L}malloc(y)}{C \vdash x \to L}$$

$$\frac{C \vdash x = y \quad C \vdash y \to t}{C \vdash x \to t} \qquad \frac{C \vdash (*x) = y \quad C \vdash y \to t \quad C \vdash x \to p}{C \vdash p \to t}$$

$$\frac{C \vdash x = y + z \quad C \vdash y \to t}{C \vdash x \to t} \qquad \frac{C \vdash x = (*y) \quad C \vdash y \to t \quad C \vdash t \to q}{C \vdash x \to q}$$

$$\frac{C \vdash call(M, x)^{L} \quad C' = C \cdot L \; M(y)\{s\} \quad C \vdash x \to t}{C' \vdash y \to t \quad C' \vdash s}$$

$$\frac{C \vdash s1; s2}{C \vdash s1 \quad C \vdash s2} \qquad \frac{C \vdash if\ (x)\ \ s1\ \ else\ s2}{C \vdash s1 \quad C \vdash s2} \qquad \frac{C \cdot L \vdash x \to t \quad isGlobal(x)}{C \vdash x \to t}$$

Example

```
A () {
1.    s=t
2:    B( )
3:    C( )
}

B () {
10:  D(&x, &y )
11:  t=x
}

C () {
20:  D(&x, &m )
21:  t=x
}

D (p,q) {
30:    (*p)=q
}
```

Context Sensitivity in Flow Sensitive Static Dependence Analysis

```
A () {
1:    B( )
2:    C( )
}
```

```
B () {
10:  x=...
11:  D ( )
12   .. = x
}
```

```
C () {
20:  x=...
21:  D( )
22:  ...= x
}
```

```
D ( ) {
30:    y=...
}
```

–How would you design the rules?

Traditional Data Flow Analysis Framework

For each block node n and every variable x
$AD^{in}[x@n]=Ad_{out}[x@n] = \emptyset$
change = true;
while change do begin
    change = false;
    for any n and x
        $AD^{in}[x@n]=\wedge_{n's\ predecessor\ n_p} AD_{out}[x@n_p]$
        oldvalue = $Ad_{out}[x@n]$;
        $Ad_{out}[x@n] = F(AD^{in}[x@n])$
        if $Ad_{out}[x@n]$!= oldvalue then change = true;
    end
end

Two important ingredients of data flow analysis

- Abstract domain
  - The results we want to compute by static analysis
- Transfer function
  - How the abstract values are computed/updated at each relevant instruction
    - Need to consider the instruction semantics

Reaching Definition Analysis

- Analyze multiple paths at a time and compute aggregate information directly.
  - $Def_{in}[x@n]$: all the possible definitions of x along some path reaching n (before getting through n)

    $Def_{in}[x@n]=\wedge_{n's\ predecessor\ n_p} Def_{out}[x@n_p]$
  - For any x!=y (node n is "y=...")

    $Def_{out}[x@n]=Def_{in}[x@n]$
  - $Def_{out}[y@n]=\{n\}$

Example for Computing Dependences

```
1  Input (x,y);
2  if (x<0)
3     p=-y;
4  else
5     p=y;
6  z=1
7  while (p!=0)
8     z=z*x
9     p=p-1;
10 Output(z);
```

Again, two important properties

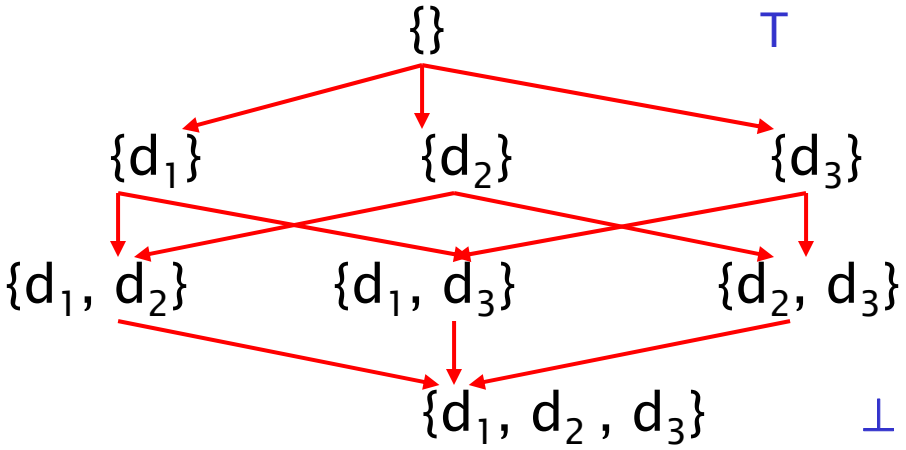
- Termination
  - Semi-lattice
- Soundness
  - Precision loss by non-distributivity

Semi-lattice

- A semi-lattice is a domain of values V and a meet operator $\wedge$ such that,
  - $\forall\ a, b, \&\ c \in V$:
    1. $a \wedge a = a$ *(idempotent)*
    2. $a \wedge b = b \wedge a$ *(commutative)*
    3. $a \wedge (b \wedge c) = (a \wedge b) \wedge c$ *(associative)*
  - $\wedge$ imposes a partial order on V, $\forall\ a, b, \&\ c \in V$:
    1. $a \geq b \Leftrightarrow a \wedge b = b$
    2. $a > b \Leftrightarrow a \geq b$ and $a \neq b$
    3. $a \geq b$ and $b \geq c$, then $a \geq c$
  - A semi-lattice has a top element, denoted T
    1. $\forall\ a \in V, a \leq T$
    2. $\forall\ a \in V, T \wedge a = a$

Semi-lattices for previous examples

- Def[x@n]: the possible definitions of x at n



-Lattice + monotonicity + finite height = termination

Lost of Precision by Directly Computing Aggregate Information

```
1  x=foo();
2  y=gee();
3  if (…)
4     p=&x;
5     q=&x;
6  else
7     p=&y;
8     q=&y;
9  *p=*q
10 *(*p)();
```

- Distributive analysis: the aggregation of individual path analysis results is equivalent to computing the aggregate information directly

  $F(a \wedge b)=F(a) \wedge F(b)$

Is the Following Analysis Distributive

- Constant propagation
  - 1 V 2 = NonC

```
1  if (…)
2     x=1;
3     y=2;
4  else
5     x=2;
6     y=1;
7  z=x+y
```

Summary

- The essence of static analysis is similar to dynamic analysis
  - Execute it without the concrete values (abstract interpretation)
  - We can express our analysis with abstract semantics just like concrete semantics
    - You can implement static analysis just like a dynamic analysis
  - Two important properties: termination and soundness
- For better scalability, we come up with different approximations
  - Merge-before-continue semantics
  - Data flow analysis
  - Flow insensitive analysis
  - Context sensitive versus context insensitive analysis