Discussion Session 9: Soufflé, Pointer Analysis

Soufflé

We now consider a simple Datalog program, and explore some of the features of Soufflé.

Say we have a Datalog file `example.dl`, whose contents are as shown:

```
.decl edge(x:number, y:number)
.input edge

.decl path(x:number, y:number)
.output path

path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

We see that `edge` is a `.input` relation, `path` is a `.output` relation.

If the input `edge` relation is pairs of vertices in a graph, by these two rules the output `path` relation will give us all pairs of vertices x and y for which a path exists in that graph from x to y.

For instance, if the contents of the tab-separated input file `edge.facts` is

```
1      2
2      3
```

The contents of the output file `path.csv`, after we evaluate this program, will be

```
1      2
2      3
1      3
```

We can evaluate this program by running

```
$ souffle -F. -D. example.dl
```

The `-F` and `-D` options specify the directories for input and output files respectively. So in this case, `edge.facts` is in the current working directory `.`, and `path.csv` will be produced here also.

Instead, if `edge.facts` was in a subdirectory called `input`, and we wanted to have `paths.csv` in a subdirectory called `output`, we could do

```
$ souffle -F./input -D./output example.dl
```

Running Soufflé in this way uses the *interpreter* mode, which means the Datalog program is evaluated immediately.

Soufflé also supports a *compiler* mode, which transforms the Datalog program into a C++ program, which is then compiled to produce an executable. Check out `-o` option.

Besides, the `-r` option is useful for debugging, as it generates a debug report in html format.

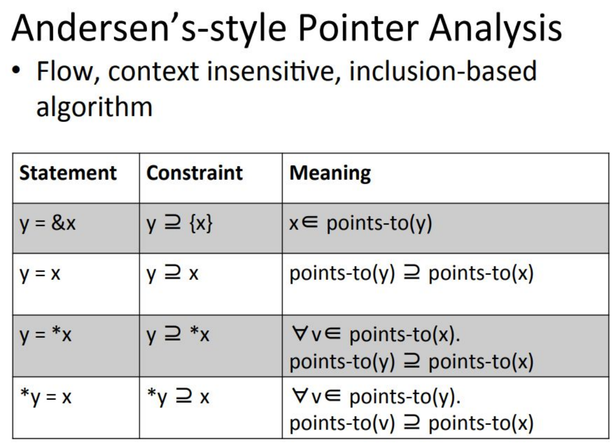
Soufflé Tutorial: https://souffle-lang.github.io/tutorial

Pointer Analysis

1. **A pointer language** (assume x and y are pointers):

- `y = &x`: y points to x
- `y = x`: if x points to z, then y points to z
- `*y = x`: if y points to z and z is a pointer, and if x points to w, then z now points to w
- `y = *x`: if x points to z and z is a pointer, and if z points to w, then y now points to w
- `points-to(x)` (or `pts(x)` from lecture slides): a set of variables that pointer variable x may point to

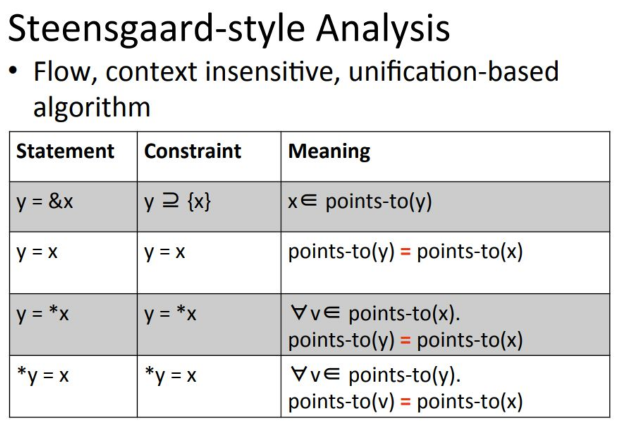
2. ***Andersen* pointer analysis vs. *Steensgaard* pointer analysis:**



Andersen’s-style Pointer Analysis

- Flow, context insensitive, inclusion-based algorithm

Statement	Constraint	Meaning
y = &x	y ⊇ {x}	x∈ points-to(y)
y = x	y ⊇ x	points-to(y) ⊇ points-to(x)
y = *x	y ⊇ *x	∀v∈ points-to(x). points-to(y) ⊇ points-to(x)
*y = x	*y ⊇ x	∀v∈ points-to(y). points-to(v) ⊇ points-to(x)



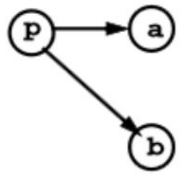
Steensgaard-style Analysis

- Flow, context insensitive, unification-based algorithm

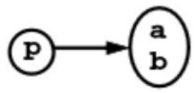
Statement	Constraint	Meaning
y = &x	y ⊇ {x}	x∈ points-to(y)
y = x	y = x	points-to(y) = points-to(x)
y = *x	y = *x	∀v∈ points-to(x). points-to(y) = points-to(x)
*y = x	*y = x	∀v∈ points-to(y). points-to(v) = points-to(x)

Here are two examples:

That is, in Andersen's Algorithm we might have

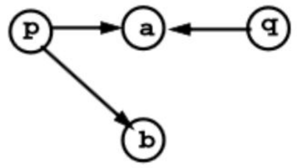


In Steensgaard's Algorithm we would instead have

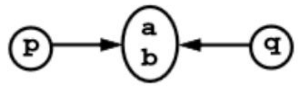


In effect any two locations that might be pointed to by the same pointer are placed in a single equivalence class.

Steensgaard's Algorithm is sometimes less accurate than Andersen's Algorithm. For example, the following points-to graph, created by Andersen's Algorithm, shows that `p` may point to `a` or `b` whereas `q` may only point to `a`:



In Steensgaard's Algorithm we get



incorrectly showing that if `p` may point to `a` or `b` then so may `q`.

3. **Summary**:

- Andersen-style: many out edges, one variable per node
- Steensgaard-style: one out edge, many variables per node

Credit:
https://courses.cs.washington.edu/courses/cse501/15sp/slides/L5-pointer-analysis.pdf
http://web.cs.iastate.edu/~weile/cs513x/2.PointerAnalysis.pdf