Static Analysis as a Path Problem
CS701

Thomas Reps
[Based on notes taken by Matt Amodio on October 8, 2015]

Abstract

This lecture introduces the notion of static-analysis problems as *path problems*. Rather than introducing the concept of a path problem at the same time as we introduce program-analysis notions, we start by discussing the shortest-distance problem.

1 Shortest-Distance Problem

The shortest-distance problem[1] is a well-studied problem in graph theory in which we try to find the shortest distance from a particular node s to every other node. Technically, this is called the *single-source shortest-distance* problem (*SSSD*). *SSSD* can be formulated in different ways, and there are different algorithms for solving it. We will give two formulations that are analogous to the formulations that we will use for dataflow-analysis problems. You should bear in mind that not all characteristics of the shortest-distance problem carry over to dataflow-analysis problems; in particular, the most efficient algorithms for the shortest-distance problem are different from the algorithms we will use for dataflow analysis.

First, let us consider an example

$$SSSD(s, v) \text{ with } v \in \{s, t, a, b, c\}$$

where all edge lengths are non-negative, and the graph is as shown below:

$$\begin{tikzcd}
 & b \arrow[d, bend left, "2"] \arrow[dr, "11"] & \\
s \arrow[ur, "3"] \arrow[r, "7"'] \arrow[dr, "8"'] & a \arrow[u, bend left, "1"] \arrow[r, "8"'] \arrow[d, "2"] & t \arrow[ul, bend right, "2"'] \\
 & c \arrow[ur, "6"'] &
\end{tikzcd}$$

[1]Sometimes called the "shortest-path problem."

In this example, we can manually figure out that

$$\begin{aligned}
SSSD(s,s) &= 0\\
SSSD(s,b) &= 3\\
SSSD(s,a) &= 5\\
SSSD(s,c) &= 7\\
SSSD(s,t) &= 13
\end{aligned}$$

2 Similarities/Differences Between Dataflow-Analysis Problems and Path Problems

2.1 Similarities

- Edge "weights" (lengths in *SSSD*; actions/transformers in dataflow analysis)
- Multiple paths to a given node
- Solvable by worklist algorithm

2.2 Differences

1. In an *SSSD* problem, the reason that a node n has a particular distance can always be justified by the distance given to one of n's predecessors m and the length of the edge (m, n).

   In contrast, in a dataflow-analysis problem, the justification for node n's value may require looking at the value of *several* of n's predecessors, $m_1, \ldots, m_k$ and the actions on the edges (m_1, n) …, (m_k, n).
2. In an *SSSD* problem, we can always identify at least one shortest path (although there may be ties). (A shortest-path can be read off by identifying a witness predecessor à la item 1, a witness predecessor of the witness predecessor, and so on.)

   In contrast, because in a dataflow-analysis problem a node may have a *set* of witness predecessors, the generalized structure that serves in place of a witness *path* has the structure of an *and-or graph* [2, §3.1.4 and §3.2.1].
3. In an *SSSD* problem, an algorithm need only "consider" a node once.

   Let me elaborate on what I mean by this observation. When all edge lengths are non-negative, Dijkstra's algorithm can be applied [1]. That algorithm may assign a *candidate distance* to a node multiple times, but that is not what I mean by "considering" a node. The one time a node n is "considered" is when n is the node selected from the worklist as one of the nodes with the shortest distance among the nodes that have yet to be "considered."

   In contrast, in worklist-based algorithms used in dataflow analysis, a given node n may be selected from the worklist—i.e., "considered"—multiple times.
4. In an *SSSD* problem with non-negative edge lengths, cycles do not matter (in the sense that paths that go through cycles are never the path that serves as the witness of a shortest distance from s to n).

3 Alternative Ways of Formulating a Path Problem

3.1 Declarative Formulation of the Desired Answer

Distance Along a Path

$$s \xrightarrow{\quad l_1 \quad} \text{op} \xrightarrow{\quad l_2 \quad} \text{op} \xrightarrow{\quad l_3 \quad} t$$

We think of "op" in the diagram above as an operator placed between l_1 and l_2 and between l_2 and l_3, giving the expression $l_1 \,\text{op}\, l_2 \,\text{op}\, l_3$. The operator "op" will be referred to generically as "extend", and denoted by $\otimes$. In the *SSSD* problem, $\otimes$ is addition ($+$), which happens to be a commutative operation. (In contrast, in dataflow-analysis problems, $\otimes$ will typically be non-commutative.)

Note that

- Each path from s to n has a finite number of edges (i.e., each path is of finite length).
- The number of paths from s to n can be infinite due to cycles.

Combining Distances from Two or More Paths

Consider the following situation:

$$\begin{tikzcd}
& {} \arrow[r, "l_2"] & {} \arrow[dr, "l_3"] & \\
s \arrow[ur, bend left, "l_1"] \arrow[dr, bend right, "l_5"] & & & \text{op } t \\
& {} \arrow[r, "l_4"] & {} \arrow[ur, "l_6"] &
\end{tikzcd}$$

The operator labeled op in the diagram above will be referred to generically as *combine*, and denoted by $\oplus$. In the *SSSD* problem, $\oplus$ is min. min is commutative, which is also the case with $\oplus$ in dataflow-analysis problems.

We can specify the solution to the shortest-distance problem as follows:

Let l : Edges $\rightarrow \mathbb{N}$ be the assignment of non-negative lengths to edges. Let p be a path $e_1, e_2, \ldots, e_k$. The *length* of p is defined as follows:

$$\text{len}(p) \overset{\text{def}}{=} l(e_1) + l(e_2) + \ldots + l(e_k).$$

Then

$$\mathit{SSSD}(s,t) = \underset{p \in \text{Paths}(s,t)}{\text{Min}} \text{len}(p).$$

Using our generic notation, the *length* of p is defined as follows:

$$\text{len}(p) \overset{\text{def}}{=} l(e_1) \otimes l(e_2) \otimes \ldots \otimes l(e_k),$$

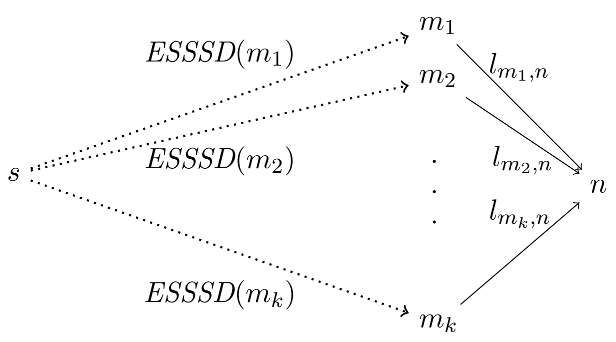
and

$$\mathit{SSSD}(s,t) = \bigoplus_{p \in \text{Paths}(s,t)} \text{len}(p). \tag{1}$$

This formulation really just serves as a specification of the problem. Its outlook is global, because is specifies the answer in terms of the global concepts of the set of all paths in $\mathit{Paths}(s,t)$. We will call the node values specified by this formulation of the problem the *combine-over-all-paths solution.*

3.2 Equational Formulation

In the *equational formulation* of a path problem, we adopt a local view. For a given node n, we look at all of its predecessors $m_1, \ldots, m_k$.



Each node n of the graph gives rise to a variable $ESSSD(n)$. We can formulate the *SSSD* problem as follows:

$$\begin{aligned} ESSSD(s) &= 0 \\ ESSSD(n) &= \underset{(m_i,n)\in \text{Edges}}{\text{Min}} ESSSD(m_i + l_{m_i,n}) \end{aligned}$$

or, using our generic notation, as

$$\begin{aligned} ESSSD(s) &= 0 \\ ESSSD(n) &= \bigoplus_{(m_i,n)\in \text{Edges}} ESSSD(m_i \otimes l_{m_i,n}) \end{aligned}$$

Equational Formulation of the Example

Returning to the running example, we have the following set of equations:

$$\begin{aligned} ESSSD(s) &= 0 \\ ESSSD(a) &= \min(ESSSD(s) + 7, ESSSD(b) + 2) \\ ESSSD(b) &= \min(ESSSD(s) + 3, ESSSD(a) + 1, ESSSD(t) + 2) \\ ESSSD(c) &= \min(ESSSD(s) + 8, ESSSD(a) + 2) \\ ESSSD(t) &= \min(ESSSD(a) + 8, ESSSD(b) + 11, ESSSD(c) + 6) \end{aligned}$$

We wish to solve the equations by a worklist algorithm that performs a process of successive approximations. Before we can get started, we have to decide how to initialize the variables—i.e., what is the initial approximation? We will set $s \mapsto 0$ and everything else to ∞. To motivate this choice, recall what we did for the binding-time analysis phase of partial evaluation, which was another worklist algorithm. There, we set dynamic variables to D and everything else to S. In both situations, we are making the analogous choice: we assume an extremal value, picking the extremum that indicates the *absence of influence*. The value only changes from that extremal value when we have proved that there could be an influence from the initial node s.

In this case, iteration by iteration we obtain

Iteration:	0	1	2	3	4
s	0	0	0	0	0
a	∞	7	5	5	5
b	∞	3	3	3	3
c	∞	8	7	7	7
t	∞	∞	14	13	13

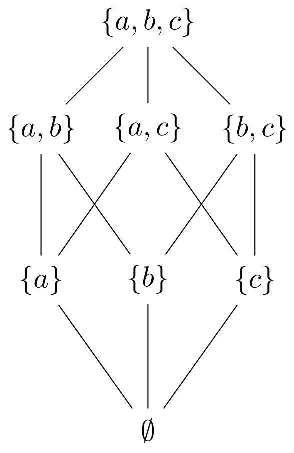
Once we have the same set of values on consecutive iterations, the successive-approximation process has quiesced—we are done!

3.3 Lattice Duality

In the *SSSD* problem, the values were totally ordered. There are two possibilities:

1. Put 0 at the bottom and ∞. In between, working up from 0, we have 1, 2, ...
2. put ∞ at the bottom and 0 at the top. In between, working down from 0, we have 1, 2, ...

In dataflow analysis, we work with value spaces that are *partially ordered.* An example of such a space is the power set of $\{a, b, c\}$, which is illustrated below.



Unfortunately, the dataflow-analysis literature and the abstract-interpretation literature orient the partially ordered sets in opposite ways. (This difference is a vestige of the two subareas having been developed mostly independently in an era in which scientific communication was more difficult than it is now.) In the dataflow-analysis community, problems are typically formulated as "meet-over-all-paths problems," while the order in the abstract-interpretation community leads to "join-over-all-paths problems."

Fortunately, it does not really matter, except that you have to orient yourself in different ways depending on which camp the author of a paper you are reading is in. All results obtainable in one orientation are obtainable in the other orientation, thanks to the *duality property of lattices.* Intuitively, just think of flipping the orientation of the diagram above: $\cup$ becomes $\cap$, $\cap$ becomes $\cup$, $\perp$ becomes $\top$, and $\top$ becomes $\perp$. In general, if property P holds in one orientation, then in the other orientation the property that holds is $P[\cup \leftarrow \cap, \cap \leftarrow \cup, \perp \leftarrow \top, \top \leftarrow \perp]$ (where $x \leftarrow y$ denotes the replacement of all instances of x by y. For instance, if

$$a \cap (b \cup c) = (a \cap b) \cup (a \cap c)$$

for one orientation, then

$$a \cup (b \cap c) = (a \cup b) \cap (a \cup c)$$

holds for the other orientation.

3.4 Dataflow Analysis

Moving back to dataflow analysis, we must make a few changes. In dataflow analysis, we do not always work with powerset lattices, so instead of $\cup$ and $\cap$, we have the *join* ($\sqcup$) and *meet* ($\sqcap$) operators.) I will generally adopt the orientation used by the abstract-interpretation community, so $\oplus$ is $\sqcup$ instead of min. For purposes of this example, I will use a powerset lattice, so $\oplus$ is $\cup$.

$$\begin{tikzcd}
\{p\} \arrow[dr] & \\
& \{p,r\} \\
\{r\} \arrow[ur] &
\end{tikzcd}$$

The $\otimes$ operator is now (the reversal of) function composition, instead of addition. Consider the path depicted below:

$$\begin{tikzcd}
\emptyset \arrow[r, "\lambda z.z \cup \{r\}"] & \cdot \arrow[r, "\lambda z.z \cup \{s\}"] & \cdot \arrow[r, "\lambda z.z \cup \{t\}"] & \cdot
\end{tikzcd}$$

The result of applying the transformation represented by this path is

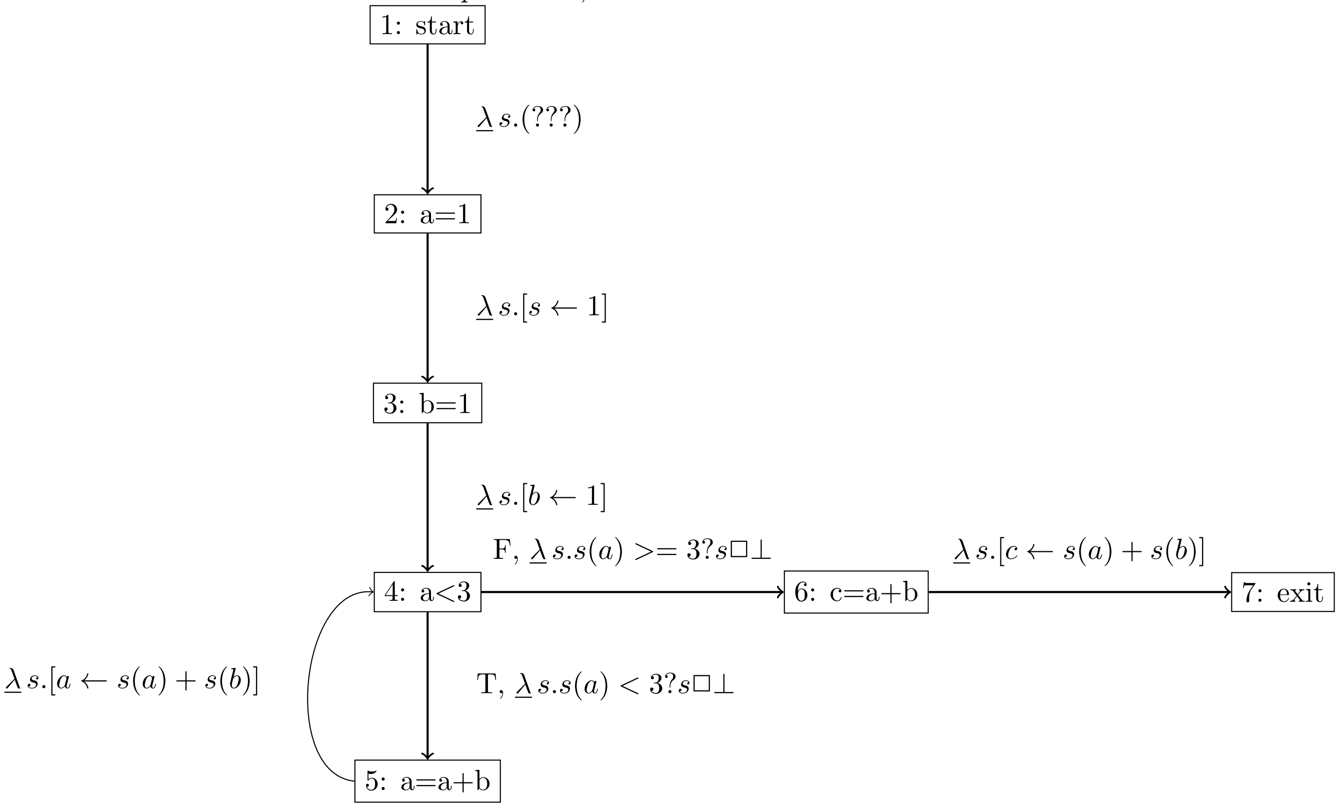
$$(\lambda z.z \cup \{t\})((\lambda z.z \cup \{s\})((\lambda z.z \cup \{r\})(\emptyset))) = \{r, s, t\}$$

Alternatively, we can consider the compound transformation performed by the path, namely, the composition from right-to-left of the three functions:

$$(\lambda z.z \cup \{t\}) \circ (\lambda z.z \cup \{s\}) \circ (\lambda z.z \cup \{r\}) = \lambda z.z \cup \{r, s, t\}.$$

3.5 An Example

We will formulate a particular dataflow-analysis problem, namely, "For each point p in the program, what variables have a constant value at p?" First, we need to define some notation.


1: start
$\underline{\lambda}\, s.(???)$
2: a=1
$\underline{\lambda}\, s.[s \leftarrow 1]$
3: b=1
$\underline{\lambda}\, s.[b \leftarrow 1]$
F, $\underline{\lambda}\, s.s(a) >= 3?s\square\bot$
$\underline{\lambda}\, s.[c \leftarrow s(a) + s(b)]$
4: a<3
6: c=a+b
7: exit
$\underline{\lambda}\, s.[a \leftarrow s(a) + s(b)]$
T, $\underline{\lambda}\, s.s(a) < 3?s\square\bot$
5: a=a+b

Some standard notation.

$$\text{Store} = (\text{Var} \to \mathbb{Z} \cup \{?\})_\perp$$

$$M : \text{Edges} \to \text{store transformer}; \text{Store}_\perp \to \text{Store}_\perp$$

$$M(<1,2>) = \lambda s.(?,?,?) = \lambda s.(a \mapsto ?, b \mapsto ?, c \mapsto ?)$$

$$\underline{\lambda}\, s.e \quad \text{means} \quad \lambda s.s = \perp ? \perp \Box e$$

$$s[a \leftarrow v] = \lambda k.k = a ? v \Box s(k)$$

Collecting semantics. Let M be the assignment of edges to actions. Let $q = e_1, e_2, \ldots, e_k$ be a path from s to n; q's *path transformer* $PT(q)$ is

$$M(e_k) \circ \ldots \circ M(e_2) \circ M(e_1).$$

Then for each node n, the *collecting semantics* at n is the set of Stores defined as follows:

$$CS(n) = \{PT(q)(\sigma) \mid p \in \text{Paths}(s,n), \sigma \in \text{Store}\} \tag{2}$$

The collecting semantics is already an abstraction, in the sense that it has forgotten all information about the order in which stores arise at n—or even whether two stores σ_1 and σ_2 can arise in the same run.

Note, however, that Eqn. (2) does not have exactly the same form as Eqn. (1). We will address this issue next time.

References

[1] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[2] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. of Comp. Prog.*, 58(1–2):206–263, October 2005.