Interprocedural Dataflow Analysis, Part I
CS701

Thomas Reps
[Based on notes taken by Sek Cheong on October 15, 2015]

Abstract

This lecture discusses interprocedural dataflow analysis. We describe the "functional approach" to interprocedural dataflow analysis pioneered by Sharir and Pnueli [6], and show some connections to the kind of path problems discussed in the previous two lectures. We will also look at the family of so-called "gen/kill problems" because they are the simplest class of problems that helps to provide some insight on some of the issues that arise in Sharir/Pnueli algorithm.

1 Pointer Analysis

In terms of connecting pieces end-to-end so that one can write a sound analyzer for a compiler or a bug-finding tool, we are omitting—for now—a key step, which is what to do about pointers. The reason for doing so is that pointer-analysis algorithms generally make use of techniques other than the path problems that I have been discussing. The method(s) for interprocedural dataflow analysis that I will present are related to path problems, and I don't want to lose the context that we have built up. However, I do want to give you an idea of the kind of information that pointer analysis supplies.

Statement Normalization. A program can contain pointer-manipulation statements like the following ones:

```
 x = *p;
*q = y;
```

What do we do about these pointer dereferences? First, the program is normalized into a form in which all assignment statements are broken down into uses of four kinds of statements:

```
 p = &x;
 p = q;
 x = *p;
*q = y;
```

Note that if we have a statement that is not of this form, such as,

```
x = ***q;
```

the analyzer will introduce temporary variables to hold on to the results of successive pointer-dereference operations, e.g.,

```
t1 = *q;
t2 = *t1;
x = *t2;
```

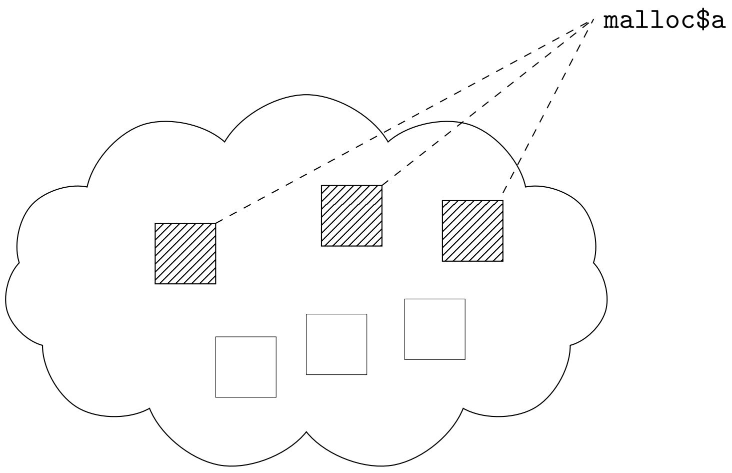


Figure 1: Depiction of a concrete heap and how its elements relate to the summary node `&malloc$a` that would be introduced to represent all the storage locations allocated by a statement such as "a: p = malloc(...)." The cloud stands for a snapshot of the concrete heap nodes—allocated at statement `a` and elsewhere. The shaded boxes represent the storage locations that were allocated at statement `a`. The white boxes represent storage locations that were allocated at statements other than `a` (which would be represented by other summary locations.

Use of Points-To Information. Pointer analysis gives us information in the form of p points to some set of variables, such as $p \mapsto \{y, z\}$. The points-to information $p \mapsto \{y, z\}$ means "p might point to y or p might point to z." In general, we can have zero or more elements in the set; sometimes we can have thousands of elements in the set (which is often due to a cascade of imprecision during pointer analysis). A pointer-analysis algorithm can be either *flow-sensitive* or *flow-insensitive*. A flow-sensitive algorithm provides points-to information that is specific to each point in the program. A flow-insensitive algorithm is less precise: the flow of control in the program is ignored, and all points-to information recovered is treated as holding for each point in the program.[1]

When the client of point-to analysis is yet another phase of dataflow analysis, the point-to information is typically used to (implicitly or explicitly) to treat a pointer-manipulation statement like $x = *p$ as a non-deterministic choice among a collection of assignment statements. For instance, when the points-to information for p is $p \mapsto \{y, z\}$, $x = *p$ is transformed to

```
if (*)
  x = y;
else
  x = z;
```

where "*" denotes an unknown Boolean value. If there are more elements in the points-to set, $x = *p$ would be transformed into a k-way non-deterministic choice.

Handling Heap-Allocated Storage. Another thing we need to know about points-to-analysis problems is what the analyzer does about an allocation statement, such as

[1]If the use of a flow-insensitive point-to algorithm sounds like a strange thing to do, think of it as being in the same spirit of type information. When we say that a variable has type τ, it must have "τ-like behavior" throughout the program. Similarly, when client code uses points-to facts obtained with a flow-insensitive algorithm, it must operate under the constraint that, e.g., $p \mapsto \{y, z\}$ holds throughout the program.

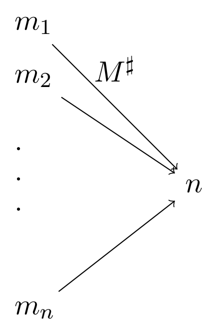


Figure 2: Equational abstract semantics.

```
    a: p = malloc(...)
```

The analyzer converts this statement into the assignment of the address of a fictitious variable `&malloc$a`:

```
    a: p = &malloc$a;
```

Here the letter `a` in `&malloc$a` matches the original label `a` of the statement. This fictitious variable summarizes all storage locations allocated at program point `p`. In Fig. 1, the cloud stands for a snapshot of the concrete heap nodes—allocated at statement `a` and elsewhere. The shaded boxes represent the storage locations that were allocated at statement `a`. The white boxes represent storage locations that were allocated at statements other than `a` (which would be represented by other summary locations.

This method is basically a simple way of finitizing the *a priori* unbounded amount of storage that might be allocated at statement `a` by mapping it to some bounded-size set of descriptors.

2 Path Problems

2.1 An Anomaly

Last lecture, there was a small anomaly in our use of the path-problem framework that I introduced and discussed in previous lectures. The goal that I had articulated was to work through a sequence of problems so that we had a unified picture of dataflow-analysis problems as path problems. We started out with the single-source shortest distance problem (*SSSD*), in terms of the operators + and min. I then introduced more abstract—but unbiased—terminology; I expressed everything in terms of two binary operators, extend ($\otimes$) and combine ($\oplus$). However, there was one place where I did not quite manage to pull off the promised grand unification: when we finally got to intraprocedural (single-procedure) dataflow analysis, the equational view was expressed as the following equation (see Fig. 2):

$$EAS[s,n] \overset{\text{def}}{=} \begin{cases} \top & \text{if } n = s \\ \bigoplus_{\langle m,n\rangle \in \text{Edges}} M^\sharp(\langle m,n\rangle)(EAS[s,m]) & \text{otherwise} \end{cases} \tag{1}$$

where s is the start node, and $\top$ is the value of AStore that represents all stores in Store.

Note that Eqn. (1) is almost of the desired form, but not quite: in particular, the second line of Eqn. (1) uses *function application* (i.e., $M^\sharp(\langle m,n\rangle)(EAS[s,m])$) rather than the *extend operation.* The reason I say that we *almost* have the desired form because extend is the reversal of function

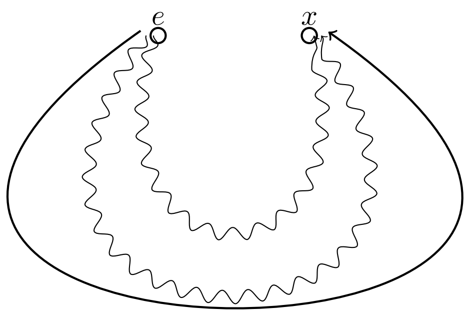


Figure 3:

composition, and function application is closely related to function composition.[2] One of the things we will do today is to fix this minor glitch.

2.2 Declarative View

Before changing the equational view, let us go back and review the declarative view for a moment. The *abstract path semantics*, denoted by $APS[s, n]$, is defined as follows:

$$APS[s,n] \stackrel{\text{def}}{=} \bigoplus_{p \in \text{Paths}(s,n)} apf_p(\top), \tag{2}$$

where the *abstract path transfer* function for path $p = e_1, e_2, \ldots, e_k$ is defined as

$$apf_p \stackrel{\text{def}}{=} M^\sharp(e_1) \otimes M^\sharp(e_2) \otimes \ldots \otimes M^\sharp(e_k),$$

and $\bigoplus$ is the combine operator of the abstract domain. Recall what is going on here: $apf_p(\top)$ is the path function associated with path p from start node s to n, applied to the abstract store that represents all possible stores that can be supplied as the initial store at s (i.e., $\top$).

Eqn. (2) handles the case of *single-procedure* dataflow analysis, and the question is how it should be generalized for the *multiple-procedure* case. Part of the machinery that we need for handling the multiple-procedure case is exactly the fix to the glitch identified in §2.1—so back to the equational view!

2.3 Equational View

Let us now return to the equational view (Eqn. (1)), and fix the glitch pointed out in §2.1. The fix is to lift Eqn. (1) from *values* to *functions*, as follows:

$$\begin{aligned} EFAS[s,n] &= Id \\ EFAS[s,n] &= \textstyle\bigoplus_{\langle m,n\rangle \in \text{Edges}} M^\sharp_{(m,n)} \circ EFAS(m), \end{aligned} \tag{3}$$

[2]The extend operation, which is the reversal of function composition, is just a slight variant of notation with which you are already familiar—namely, think of extend as the UNIX pipe operator! As we go along a path, we have a collection of operators that we successively pipe into each other:

$$\text{weight}(e_1) \otimes \text{weight}(e_2) \otimes \ldots \otimes \text{weight}(e_k) = \text{weight}(e_1) \mid \text{weight}(e_2) \mid \ldots \mid \text{weight}(e_k).$$

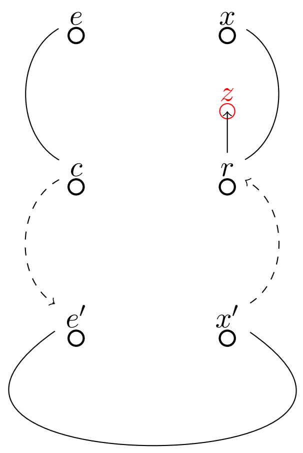


Figure 4: Interprocedural control-flow graph (ICFG).

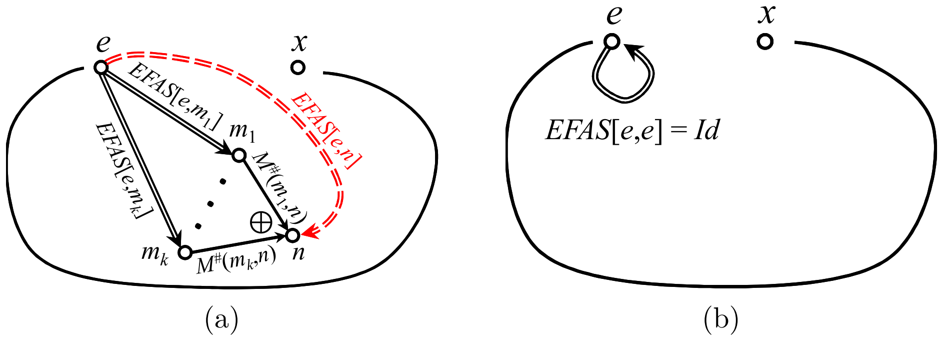


Figure 5: Depiction of the two cases in Eqn. (1).

Eqn. (3) will be explained in more detail below.

For the single-procedure case, a CFG looks like the graph depicted in Fig. 3. I'll use e to denote what we have been calling the start node. (In the multiple-procedure case, each procedure's CFG will have an *entry node*, whence the use of "e".) We use x to represent the CFG's *exit* node. In the graphical notation used in Fig. 3, a squiggly line from e to x will denote a path—or sometimes a set of paths—that start at 3 and proceed through the procedure to x.

The graph shown in Fig. 4 depicts the simplest multiple-procedure case: a caller calling a single callee. We call such a graph an *interprocedural control-flow graph*, or *ICFG*. We split each call site in the caller into two separate and distinct nodes: *call-node* c represents the call; *return-node* r represents where control returns to in the caller after the callee finishes execution. e' is the entry node of the callee and x' is the exit node of the callee.

Node r represents an artificial "landing pad" for transferring control back to the caller. At the machine-code level, the callee returns control to the instruction that follows the call instruction in the caller. Our diagrams are an idealized model; the node that represents the first real action in the caller after control returns from the call will be modeled by the successor of r (i.e., node z in Fig. 4).

The edge (c, e') is called a *linkage-in* edge; the edge (x', r) is called a *linkage-out* edge.

Eqn. (3) can be understood using two pictures, as shown in Fig. 5. Fig. 5(a) looks similar to the picture used in the Oct. 13, 2015 lecture to illustrate the equational formulation of the *SSSD* problem. In Fig. 5(a), we have k predecessors of node n, $m_1, \ldots, m_k$, and the edges from each m_i to n is labeled with a transformer function $M^\sharp(m_i, n)$. Each (solid black) double-edged arrow represents an *EFAS* value, which itself is a transformer function. This information is used in Eqn. (3) to create the function that represents the transformation from e to n by composing appropriate pairs of $M^\sharp$ and *EFAS* functions and combining the results:

$$\mathit{EFAS}[s,n] = \bigoplus_{\langle m,n\rangle \in \text{Edges}} {M^\sharp}_{(m,n)} \circ \mathit{EFAS}(m).$$

The result is represented in Fig. 5(a) by the (dashed red) double-edged arrow from e to n.

Fig. 5(b) shows how the process of creating *EFAS* values (i.e., double-ruled lines) gets started, namely, we start with a single double-ruled line from e to e, which represents the 0-length path from e to itself. The transformation associated with this path is the identity transformation, denoted by *Id*.

Note how this approach is a different method for intraprocedural dataflow analysis. Earlier, we performed a successive-approximation method on *dataflow values*; in contrast, Eqn. (3) and Fig. 5 perform a successive-approximation method on *dataflow transformers*.

Finally, because compose is the reverse of extend, we can rewrite Eqn. (3) as follows, using the extend operator:

$$\begin{array}{rcl} \mathit{EFAS}[s,n] & = & \mathit{Id} \\ \mathit{EFAS}[s,n] & = & \bigoplus_{\langle m,n\rangle \in \text{Edges}} \mathit{EFAS}(m) \otimes M^\sharp_{(m,n)}. \end{array} \tag{4}$$

Note how Eqn. (4) fixes up the anomaly discussed in §2.1; Eqn. (4) is now of the desired form for a path problem!

It must be pointed out that there is just a small amount of sleight-of-hand here. In Eqn. (1), the combine operation ($\oplus$) is a binary operation on *dataflow values*, whereas in Eqns. (3) and (4) $\oplus$ is now a binary operation on *dataflow transformers* (i.e., functions from dataflow values to dataflow values).

What do we mean by a "combine operation on functions?" Basically, the function $f_1 \oplus f_2$ constructed by combining two functions f_1 and f_2 maps a value in the domain-space into two values in the range-space, and takes the combine of those values in the range-space:

$$(f_1 \oplus f_2)(v) = f_1(v) \oplus f_2(v).$$

For instance, suppose that f_1 and f_2 operate on sets:

$$f_1 \stackrel{\text{def}}{=} \lambda S.S \cup \{a, b\}$$

and

$$f_2 \stackrel{\text{def}}{=} \lambda S.S \cup \{b, c\}.$$

Then, assuming that the combine operation on values is set-union, we have

$$(f_1 \oplus f_2)(S) = f_1(S) \oplus f_2(S) = S \cup \{a, b, c\}.$$

3 Gen/Kill Problems

First of all, it probably would have been better if this class of dataflow-analysis problems were called kill/gen problems, rather than gen/kill problems. As we will see, the order in which operations occur in the dataflow transformers is that a "kill" operation comes first, followed by a "gen" operation. No

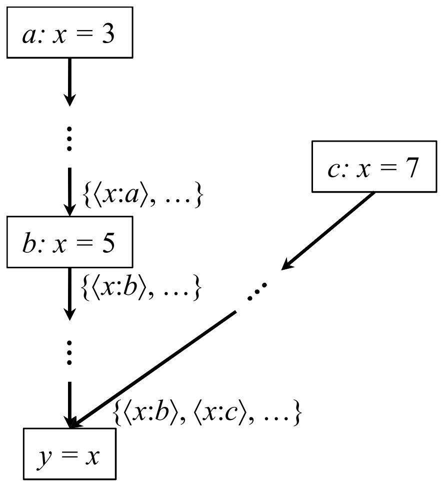


Figure 6: An example to illustrate the reaching-definitions problem.

matter—it is conventional to call them *gen/kill problems*. Many simple dataflow-analysis problems are expressible by gen/kill functions, which is where this family of problems gets their name. To be concrete, we will illustrate one problem that is useful for static taint analysis, namely, the *reaching-definitions problem*.

Consider the example shown in Fig. 6. There are several statements that assign to variable x—those at labels a, b, and c. There is also a use of x in the statement $y = x$ shown at the bottom. Our goal is to gather up information so that what we know at a node n reflects what may hold just before n executes. In this case, we want to know the answer to the question, "For each node n, what are the program points that might have assigned into x the value that x holds just before n executes?" (Actually, we want to know the answer to this question for each of the program variables in scope at each node n.)

Here is how we can formalize this problem as a gen/kill problem. In the dataflow transformer for program-point p, the input set S is transformed by (i) subtract some set $Kill_p$ from S, and (ii) union the result of (i) with some other set Gen_p. In other words, each dataflow transformer has the form

$$\lambda S.(S - Kill_p) \cup Gen_p,$$

where $Kill_p$ and Gen_p are constant sets associated with program-point p. With different points we can have different functions. The universe of gen/kill functions is

$$F \stackrel{\text{def}}{=} \{\lambda S.(S - K) \cup G \mid K \subseteq A, G \subseteq A\},$$

where A is some universe of dataflow facts. Note that A typically depends on the program being analyzed, so it would be more precise to refer to A_{prog} rather than A. We will leave off the subscript prog, except for emphasis.

In the case of reaching-definitions, A_{prog} is defined as follows:[3]

$$A_{\text{prog}} \stackrel{\text{def}}{=} \{\langle v : l\rangle \mid \text{variable } v \text{ is assigned to at } l\}.$$

[3]When there are fictitious variables of the form `&malloc$a` introduced for the purpose of points-to analysis in the presence of heap-allocated storage, we need to say $A_{\text{prog}} \stackrel{\text{def}}{=} \{\langle v : l\rangle \mid \text{variable } v \textit{ may} \text{ be assigned to at } l\}$.

Note that in the absence of heap-allocated storage,

$$|A_{\text{prog}}| \approx \text{number of nodes in the CFG.}$$

Returning to Fig. 6, at node b, the kill set $Kill_b$ is

$$K_b \overset{\text{def}}{=} \{\langle x : l\rangle | x \text{ is assigned to at } l\};$$

the gen set Gen_b is $\{\langle x : b\rangle\}$; and the dataflow transformer at b is $\lambda S.(S - Kill_b) \cup Gen_b$. Note that $Kill_b$ actually removes $\langle x : b\rangle$ from S; however, that is OK because Gen_b unions $\{\langle x : b\rangle\}$ into the answer set.

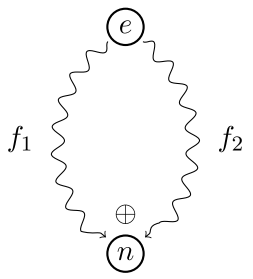
In terms of a representation scheme for gen/kill functions, a gen/kill function $\lambda S.(S - K) \cup G$ can be represented by a pair of sets $\langle K, G\rangle$. In §3.1 and §3.2, we show how to perform combine and extend using such function representations.

3.1 Combine

Suppose that we have two functions $f_1 \overset{\text{def}}{=} \lambda S.(S - K_1) \cup G_1$ and $f_2 \overset{\text{def}}{=} \lambda T.(T - K_2) \cup G_2$, represented by the pairs of sets $\langle K_1, G_1\rangle$ and $\langle K_2, G_2\rangle$, respectively. Then we want to determine the appropriate function of K_1, G_1, K_2, and G_2 that should be performed to create the representation of the function

$$f_3 = (\lambda S.(S - K_1) \cup G_1) \oplus (\lambda S.(S - K_2) \cup G_2).$$

The picture looks like the following:



After entering a procedure at e and reaching node n, what can we guarantee has been killed? Only items that have been killed along *both* paths! Consequently, the kill set for f_3 is $K_1 \cap K_2$. Similarly, the gen set for f_3 consists of items that have been gen'd along *either* path, so the gen set for f_3 is $G_1 \cup G_2$.

Thus, when each gen/kill function $\lambda S.(S - K) \cup G$ is represented by a pair of sets $\langle K, G\rangle$, we can perform combine as follows:

$$\langle K_1, G_1\rangle \oplus \langle K_2, G_2\rangle = \langle K_1 \cap K_2, G_1 \cup G_2\rangle.$$

3.2 Extend

Suppose that we again have two functions $f_1 = \lambda S.(S - K_1) \cup G_1$ and $f_2 = \lambda T.(T - K_2) \cup G_2$, represented by the pairs of sets $\langle K_1, G_1\rangle$ and $\langle K_2, G_2\rangle$, respectively. We now want to determine the appropriate function of K_1, G_1, K_2, and G_2 that should be performed to create the representation of the function

$$f_4 = \lambda S.(S - K_1) \cup G_1 \otimes \lambda T.(T - K_2) \cup G_2. \tag{5}$$

First, we know that extend should be compose in reverse, so we start by writing Eqn. (5) as a composition.

$$\begin{aligned}
f_4 &= (\lambda T.(T - K_2) \cup G_2) \circ (\lambda S.(S - K_1) \cup G_1) \\
&= \lambda U.(\lambda T.(T - K_2) \cup G_2)((\lambda S.(S - K_1) \cup G_1)(U)) \\
&= \lambda U.(\lambda T.(T - K_2) \cup G_2)((U - K_1) \cup G_1) \\
&= \lambda U.(((U - K_1) \cup G_1) - K_2) \cup G_2 \\
&= \lambda U.((U - K_1) - K_2) \cup (G_1 - K_2) \cup G_2 \\
&= \lambda U.((U - (K_1 \cup K_2)) \cup ((G_1 - K_2) \cup G_2)
\end{aligned}$$

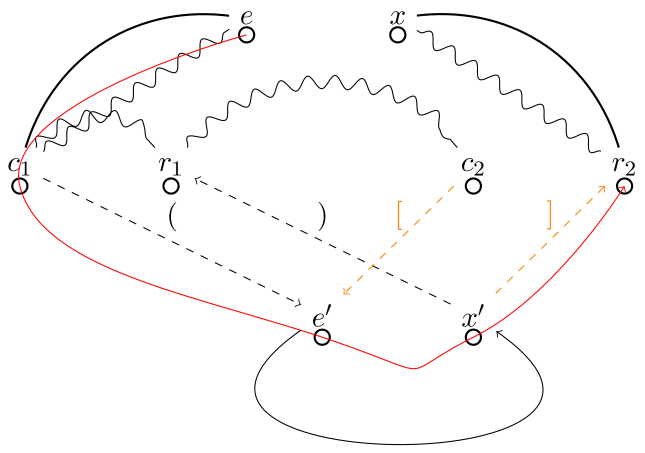
Consequently, when each gen/kill function $\lambda S.(S - K) \cup G$ is represented by a pair of sets $\langle K, G\rangle$, we can perform extend as follows:

$$\langle K_1, G_1\rangle \otimes \langle K_2, G_2\rangle = \langle K_1 \cup K_2, (G_1 - K_2) \cup G_2\rangle.$$

4 The Functional Approach to Interprocedural Dataflow Analysis

The equations for *EFAS* given in Eqn. (3) are similar to the Sharir/Pnueli equations [6], but only for the intraprocedural-analysis part; we have not yet seen the interprocedural-analysis part.

Let's consider the diagram below.



Sharir and Pnueli have the notion of *IVP* (which stands for "Interprocedurally Valid Paths"). One way to understand *IVP* is that it just involves a filtering mechanism on paths, based on a simple context-free language. The way to think about it is that for every call site, we introduce a pair of matched parentheses. In our diagram, the open parenthesis is placed on the linkage-in edge, and the matching closed parenthesis is placed on the matching linkage-out edge. The idea is that if we follow a call (path in red), and we go into a called procedure via a linkage-in edge labeled "(," then we had better go back to the caller along the linkage-out edge labeled ")." You would never see a correctly executing procedure that follows a path like the red path in the diagram that has a *mismatched* set of parenthesis, namely, "... (...]."

The use of the parenthesis labels can be viewed as a policy language. More precisely, given a language L whose alphabet consists of the labels on the edges of the graph, when we define the path problem. We can only include the paths for which the word spelled out as we go along the path belongs to language L. A path between two nodes does not count for anything if the path's word is not in L.

Going back to the declarative view of dataflow analysis that was given in Eqn. (2), namely,

$$APS[s,n] \overset{\text{def}}{=} \bigoplus_{p \in \text{Paths}(s,n)} apf_p(\top),$$

to specify the desired set of paths in the ICFG for *interprocedural* dataflow analysis, we merely add the path-language restriction "$p \in unbalLeft(s,n)$" to the index on the $\bigoplus$ operator

$$APS[s,n] \overset{\text{def}}{=} \bigoplus_{p \in unbalLeft(s,n)} apf_p(\top).$$

We now describe the language *unbalLeft* of *unbalanced-left paths*. To do so, we will need an auxiliary language of *matched* paths defined by the following context-free grammar:

$$\begin{array}{rcll} matched & ::= & \epsilon & \\ & | & matched\ \ e & \\ & | & matched\ \ (_i\ \ matched\ \ )_i & \text{for } i \in CallSites \end{array}$$

We now define *unbalLeft* by the following grammar:[4]

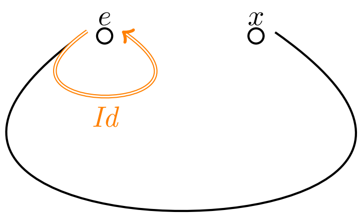
$$\begin{array}{rcll} unbalLeft & ::= & \epsilon & \\ & | & unbalLeft\ \ matched\ \ (_i & \text{for } i \in CallSites \\ & | & unbalLeft\ \ matched & \end{array}$$

5 The Sharir/Pnueli Equations in Pictures

The three pictures that follow depict the three cases in the Sharir/Pnueli equations. The first two, Eqns. (6) and (7), are essentially those for *EFAS*; the third handles cross-procedure propagation of dataflow information.

1.

$$\phi_{e,e} = Id \tag{6}$$



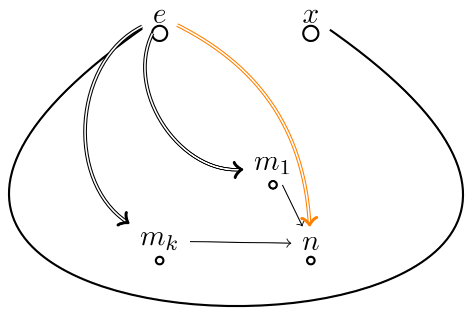
2.

$$\phi_{e,n} = \bigoplus_{\langle m,n\rangle \in \text{Edges}} f_{m,n} \circ \phi_{e,m} \text{ if } n \notin Ret \tag{7}$$

Alternatively, Eqn. (7) can be stated in terms of extend:

$$\phi_{e,n} = \bigoplus_{\langle m,n\rangle \in \text{Edges}} \phi_{e,m} \otimes f_{m,n} \text{ if } n \notin Ret \tag{8}$$

[4]It is possible to write other grammars for the languages $L(matched)$ and $L(unbalLeft)$. However, as we will see shortly, the ones given above have the advantage that they reflect the structure of the Sharir/Pnueli equations (and hence the structure of the Sharir/Pnueli algorithm for interprocedural dataflow analysis).

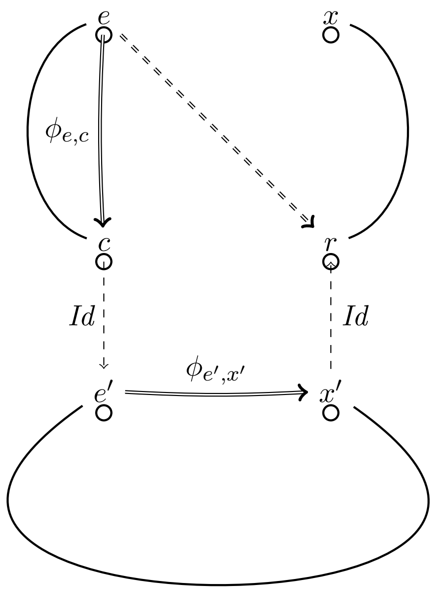


3. The third case handles cross-procedure propagation of *summary functions* of the form $\phi_{e',x'}$. Note that such a ϕ function summarizes the net effect of all matched-parenthesis paths from entry node e' to exit node x'. Sharir and Pnueli give an equation that—when re-expressed to use extend rather than function composition—looks like

$$\phi_{e,r} = \phi_{e,c} \otimes \phi_{e',x'}. \tag{9}$$

Eqn. (9) looks a bit odd because Sharir and Pnueli make the assumption that the action on every linkage-in edge and every linkage-out edge is *Id*. Thus, when accompanied by the diagram below, the equation might be more clearly expressed with two redundant occurrences of *Id*.

$$\phi_{e,r} = \phi_{e,c} \otimes \mathit{Id} \otimes \phi_{e',x'} \otimes \mathit{Id}, \tag{10}$$



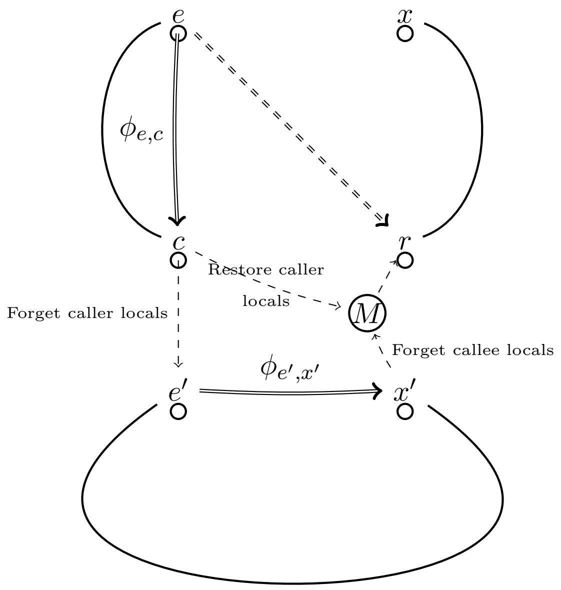
To allow the actions on linkage-in and linkage-out edges to be other than *Id*, we would use the following equation:

$$\phi_{e,r} = \phi_{e,c} \otimes \mathit{CallIn}_{c,e'} \otimes \phi_{e',x'} \otimes \mathit{CallOut}_{x',r}. \tag{11}$$

It is instructive to go back to the grammar for the language $L(\mathit{matched})$ of matched parentheses and see how the structure of Eqns. (6), (8), and (11) matches the structure of the different grammar rules exactly

$$\begin{array}{rll} \textit{matched} ::= & \epsilon & \text{(cf. Eqn. (6))} \\ | & \textit{matched} \;\; e & \text{(cf. Eqn. (8))} \\ | & \textit{matched} \;\; (_i \;\; \textit{matched} \;\;)_i & \text{(cf. Eqn. (11))} \end{array}$$

Supporting Local Variables. Although we will not cover it here, for an analysis that tracks information about local variables, and for a programming language that supports recursion, the action on the linkage-in edge is to forget information about caller locals, and at the exit node a two-argument *merge function* [3] is applied that uses (i) information from $\phi_{e',x'}$ to forget information about callee locals and (ii) information from $\phi_{e,c}$ to restore information about caller locals.



The equation that would be used in place of Eqn. (9) (or Eqn. (11)) is as follows:

$$\phi_{e,r} \quad = \quad M(\phi_{e,c}, \phi_{e',x'}).$$

Other work that uses merge functions includes [5, 4]. The idea of "peeking back at the call-site" has been imported into automata theory in the work on visibly pushdown automata [1] and nested-word automata [2].

References

[1] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, 2004.

[2] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Lang. Theory*, 2006.

[3] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Comp. Construct.*, pages 125–140, 1992.

[4] A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *Computer Aided Verif.*, 2005.

[5] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Princ. of Prog. Lang.*, 2004.

[6] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.