Data Flow Frequency Analysis

G. Ramalingam
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY, 10598, USA
rama@watson.ibm.com

Abstract

Conventional dataflow analysis computes information about what facts may or will not hold during the execution of a program. Sometimes it is useful, for program optimization, to know *how often* or with *what probability* a fact holds true during program execution. In this paper, we provide a precise formulation of this problem for a large class of dataflow problems — the class of finite bi-distributive subset problems. We show how it can be reduced to a generalization of the standard dataflow analysis problem, one that requires a sum-over-all-paths quantity instead of the usual meet-over-all-paths quantity. We show that Kildall's result expressing the meet-over-all-paths value as a maximal-fixed-point carries over to the generalized setting. We then outline ways to adapt the standard dataflow analysis algorithms to solve this generalized problem, both in the intraprocedural and the interprocedural case.

1 Introduction

Conventional dataflow analysis computes information about what facts may or will not hold during the execution of a program. An example is "type analysis", which answers questions such as whether a particular expression may evaluate to a value of a particular type at run time. Sometimes it is useful, for program optimization, to know *how often* or with *what probability* a fact holds true during program execution. Thus, for example, useful optimizations can be performed if it is known that an expression evaluates to a value of a particular type with high probability or high frequency ([6]).

The probability that a certain branch in the program will be taken at run time can be estimated either statically or using profiling information [14, 23, 3]. In this paper, we show how the frequency or probability of dataflow facts can be computed for a large class of dataflow problems, a class we call finite bi-distributive subset (FBS) problems, once every edge in the control-flow graph is annotated with a probability.

To appear in the Proceedings of the ACM SIGPLAN '96 Conference on Programming Language and Design and Implementation, Philadelphia, PA, May, 1996.

Finite bi-distributivity requires that the set of dataflow facts be finite and that the dataflow functions distribute over both set union and set intersection. This class of dataflow analysis problems is more general than the class of separable problems (also known as gen/kill or bit-vector problems) but less general than the class of finite distributive subset problems introduced by Reps et al [17].

The problem of computing the expected frequency of execution of the various statements in a program, given a control-flow graph whose edges are labelled with a probability, has been studied previously [16, 23]. We are not aware of any previous work in generalizing these techniques to general dataflow analysis problems. We provide a theoretical foundation for *dataflow frequency analysis* and outline algorithms for the problem.

The paper is organized as follows. In Section 2 we present an overview of the lattice theoretic framework for dataflow analysis and the class of finite distributive subset problems. In Section 3 we provide a precise formulation of the problem studied in this paper. We show how it can be reduced to a generalization of the standard dataflow analysis problem, one that requires a sum-over-all-paths quantity instead of the usual meet-over-all-paths quantity. In Section 4 we show that Kildall's result expressing the meet-over-all-paths value as a maximal-fixed-point carries over to the generalized setting. In Section 5 we show how this result can be utilized to compute the required solution using the standard elimination techniques for dataflow analysis. We also reinterpret the problem in terms of the exploded control-flow graph introduced by Reps et al. and show that the problem reduces to a fairly well-known *Markov chain* problem on the exploded graph (with one minor difference). In Section 6 we show how to extend the results to handle procedures. Section 7 presents our conclusion.

2 Preliminary Material

We first review the lattice-theoretic framework for dataflow analysis developed by Kildall [10].

A monotonic dataflow analysis problem is a tuple $(L, \sqcap, F, c, G, r, M)$, where:

- L is a semilattice with meet operation $\sqcap$

- $F \subseteq L \rightarrow L$ is a set of monotonic functions from L to L
- G = (V,E) is a control-flow graph with entry vertex r
- $c \in L$ is the "dataflow fact" associated with the entry vertex.
- $M : E \rightarrow F$ is a map from G's edges to dataflow functions

(For convenience, we assume that the entry vertex has no predecessors. We will also ignore procedures until Section 6.)

The function M can be extended to map every path in the graph to a function from L to L: if p is a path [e_1, e_2 ,..., e_k] then $M(p)$ is defined to be $M(e_k) \circ M(e_{k-1}) \cdots M(e_1)$. ($M([])$ is the identity function.) Given a path p from the entry vertex r to some vertex u, we define $state(p)$ to be $M(p)(c)$. The meet-over-all-paths (MOP) solution $S(u)$ to the dataflow analysis problem is defined as follows:

$$S(u) =_{def} \bigsqcap_{p \in Paths(r,u)} state(p)$$

Here $Paths(r,u)$ denotes the set of all paths p from r to u. Kildall showed that the MOP solution is the same as the maximal fixed point (MFP) of the following collection of equations:

$$\begin{aligned} x_u &= \bigsqcap_{v \rightarrow u} M(v \rightarrow u)(x_v) \ \text{ for every } u \in V - \{r\} \\ x_r &= c \end{aligned}$$

provided that every $M(e)$ is distributive. [1]

The starting point for our work is a restricted version of the monotonic dataflow analysis problem: a finite distributive dataflow analysis problem is a dataflow analysis problem where the semilattice L is the powerset 2^D of a finite set D, $F \subseteq 2^D \rightarrow_d 2^D$, the set of *distributive* functions from 2^D to 2^D, and the meet operation is set union. (This is essentially the intraprocedural version of the framework introduced by Reps et al. [17]. Analysis problems over 2^D whose meet operation is set intersection can be transformed into a set union problem by considering its dual, as noted in [17].)

Essentially, D is a (finite) set of facts that may or may not hold true at various program points and the problem is to determine which facts hold true at each program point. Occasionally, it will be convenient to think of 2^D not as the powerset of D, but as the set of functions from D to $\{0,1\}$ (eg., as a "bitvector"). Thus, if the meet-over-all-paths solution at a given program point is σ, then $\sigma(d)$ is 1 if fact d may hold at that program point, and $\sigma(d)$ is 0 if d will not hold at that program point. We will treat $\sigma \in 2^D$ as a subset of D in some places and as a function from D to $\{0,1\}$ in some places. (In other words, we will use a set and its characteristic function interchangeably.) The context should make the usage clear.

As an example, consider the reaching definitions problem. D will, in this case, be the set of definitions (statements that assign some value to some variable) in the program. If σ is the solution for a particular program point, then $\sigma(d)$ will be 1 if definition d may reach that program point, and 0 otherwise.

[1] A function $f : L \rightarrow L$ is said to be distributive if $f(\sigma_1 \sqcap \sigma_2) = f(\sigma_1) \sqcap f(\sigma_2)$. Strictly speaking, Kildall's result requires that every dataflow function f distribute over arbitrary meets: $f(\sqcap_{\sigma \in S} \sigma) = \sqcap_{\sigma \in S} f(\sigma)$. In particular, when S is empty, the meet over the empty set is $\top$, the top element of the semi-lattice, and the distributivity condition requires that $f(\top) = \top$. The usual gen/kill functions do not satisfy this. Luckily, the requirement that $f(\top) = \top$ is relevant only when there are vertices in the graph that are unreachable from entry vertex. Hence, this is not a particularly significant problem.

3 The Expected Frequency of Dataflow Facts

We are interested in extending the dataflow analysis to determine not just whether some fact may or may not hold at a program point, but also the probability that the fact may hold true at a given point (or the number of times the fact is expected to hold true at the given point). Assume that every edge e in the control-flow graph is associated with a probability $prob(e)$. We interpret $prob(u \rightarrow v)$ as the probability that execution will follow the edge $u \rightarrow v$ once it reaches u [2]. Obviously, the sum of the probabilities over all outgoing edges must be 1 for every vertex except the exit vertex, which has no outgoing edge. (Note that we do not consider *how* these probabilities are obtained. These may be estimated using various heuristics or may be based on profile data. See Section 7.)

Consider a path q from the entry vertex to some vertex u. The product of $prob(e)$ over all edges e in path q, which we will denote $prob(q)$, is the probability that program execution will follow the path q. Recall that $state(q)$ denotes (an abstraction of) the state at u after execution along this path. Let us now combine these information to identify the the number of times the various facts are expected to hold true at a given point over a single execution of the program.

Definition 1

$$\begin{aligned} E(u) &=_{def} \sum_{q \in Paths(r,u)} prob(q) \\ E(u,d) &=_{def} \sum_{\substack{q \in Paths(r,u) \\ state(q)(d)=1}} prob(q) \end{aligned}$$

$E(u)$ denotes the number of times a vertex u is expected to execute during program execution, while $E(u,d)$ denotes the number of times fact d is expected to hold true at program point u. Recall that $state(q)(d)$ is 1 if d holds true in $state(q)$ and 0 otherwise, (Thus, $E(u,d)/E(u)$ gives us the probability that d holds true at u.)

[2] An essential assumption made here is that the probability of the program execution following a particular branch is independent of the execution history, which obviously does not hold true in reality. This assumption is, however, necessary to get a handle on the problem.

Note: For any path q and vertex u, let $occurs(u, q)$ denote the number of occurrences of u in q. Similarly, let $holds(d, u, q)$ denote the number of occurrences of u in q where d holds true (during abstract execution with respect to the dataflow analysis framework under consideration). Let $exit$ denote the exit vertex of the control-flow graph. One might argue that the "expected values" should be defined as follows:

$$\begin{aligned} E(u) &= \sum_{q \in Paths(r, exit)} prob(q).occurs(u, q) \\ E(u, d) &= \sum_{q \in Paths(r, exit)} prob(q).holds(d, u, q) \end{aligned}$$

It can be shown that the above definition yields the same values as Definition 1 for every u for which *there exists a path from* u *to* $exit$ *whose probability is non-zero*. The above definition, however, does not take non-terminating program execution into account, and it is somewhat complicated to extend the definition to handle such cases. Definition 1, however, handles such cases also correctly, which is why we chose that definition.

Now, we are interested in computing the quantities $E(u)$ and $E(u, d)$ for every u and d. Our first step will be to reformulate the problem using a framework that is like a dataflow analysis framework, with one significant difference.

Let $\Re_+$ denote the set of non-negative reals, extended with ∞. Let D_Λ denote the set D extended by a new element Λ not in D. Let $\Re_+^{D_\Lambda}$ denote maps from D_Λ to $\Re_+$. The set $\Re_+^{D_\Lambda}$ is the domain of values we are interested in. (As explained above, we are interested in computing $|D| + 1$ values at every vertex u, namely $E(u)$ and $E(u, d)$ for every $d \in D$. Elements of $\Re_+^{D_\Lambda}$ essentially combine these $|D| + 1$ values into a vector.)

Our domain of values $\Re_+^{D_\Lambda}$ is not a semilattice. Instead of the usual meet operation, we will use a non-idempotent sum operation. We define the summation operator $+$: $\Re_+^{D_\Lambda} \times \Re_+^{D_\Lambda} \rightarrow \Re_+^{D_\Lambda}$ as follows:

$$(\sigma_1 + \sigma_2)(d) =_{def} \sigma_1(d) + \sigma_2(d)$$

This definition is extended in a straightforward fashion to infinite sums:

$$(\sum_{i \in I} \sigma_i)(d) =_{def} \sum_{i \in I} (\sigma_i(d))$$

for any (possibly infinite) index set I. The infinite sum $\sum_{i \in I}(\sigma_i(d))$ is well-defined since we are dealing only with non-negative numbers. If $\sum_{i \in I}(\sigma_i(d))$ does not converge to a finite value, then its value is ∞.

We define the scalar multiplication of maps from D_Λ to $\Re_+$ by a non-negative real number α ($\cdot : \Re_+ \times \Re_+^{D_\Lambda} \rightarrow \Re_+^{D_\Lambda}$) as follows:

$$(\alpha.\sigma)(d) =_{def} \alpha.(\sigma(d))$$

The only tricky part here is defining $0.\infty$. We define $0.\infty$ to be 0. This turns out to be the right definition for us, since it guarantees that multiplication by 0 is a "continuous" function: $0.(\sum_{i \in I} x_i) = \sum_{i \in I}(0.x_i)$. For any non-zero α, $\alpha.\infty$ is ∞.

We "inject" values from 2^D into $\Re_+^{D_\Lambda}$ as follows: given $\sigma \in 2^D$, we define $\widehat{\sigma} \in 2^{D_\Lambda} \subset \Re_+^{D_\Lambda}$ by:

$$\begin{aligned} \widehat{\sigma}(\Lambda) &=_{def} 1 \\ \widehat{\sigma}(d) &=_{def} \sigma(d) \quad \text{for } d \in D \end{aligned}$$

For typographical reasons, we will also denote the injection function by i: thus, $i(\sigma) =_{def} \widehat{\sigma}$. (Note that $\widehat{\sigma}$ can also be thought of as a subset of of D_Λ, namely $\sigma \cup \{\Lambda\}$.)

Now consider the following "sum over all paths" quantity:

Definition 2

$$F(u) =_{def} \sum_{q \in Paths(r, u)} prob(q).\widehat{state}(q)$$

The above definition essentially combines the $|D| + 1$ scalar quantities defined for every vertex u by Definition 1 into one vector quantity, as the following lemma shows.

Lemma 1

$$\begin{aligned} F(u)(\Lambda) &= E(u) \\ F(u)(d) &= E(u, d) \quad \text{for } d \in D \end{aligned}$$

Proof See Appendix. □

In standard dataflow analysis, we deal with the domain 2^D — the values to be computed at every vertex belong to 2^D and the transfer functions associated with the edges of the graph belong to $2^D \rightarrow 2^D$. Here, we are trying to switch to the domain of $\Re_+^{D_\Lambda}$. We have seen how we can inject a value σ from our original dataflow lattice into the value $\widehat{\sigma}$ into the new pseudo-lattice of interest. Now, let us see how to inject a transfer function $\tau \in 2^D \rightarrow_d 2^D$ into a corresponding transfer function $\widetilde{\tau} \in \Re_+^{D_\Lambda} \rightarrow \Re_+^{D_\Lambda}$. We will do this in stages.

Consider a distributive function $\tau \in 2^D \rightarrow 2^D$. The function τ is completely determined by $\tau(\phi)$ and $\{\, \tau(d) \mid d \in D \,\}$. Define $\tau^* : D_\Lambda \rightarrow 2^{D_\Lambda} \subset \Re_+^{D_\Lambda}$ by:

$$\begin{aligned} \tau^*(\Lambda) &=_{def} \widehat{\tau(\phi)} \\ \tau^*(d) &=_{def} \tau(\{d\}) - \tau(\phi) \end{aligned}$$

The above definition may be understood easiest by interpreting $\tau(\{d\})$ and $\tau(\phi)$ as subsets of D_Λ and $-$ as the set difference operator.

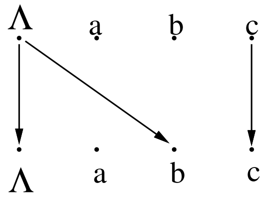
The function τ^* is essentially the *representation relation* of τ defined by Reps et al. (See [17].) The representation relation of τ, denoted by R_τ, is a binary relation on D_Λ. We can express R_τ in terms of τ^* as follows:

$$R_\tau = \{(d_1, d_2) \mid d_1 \in D_\Lambda, d_2 \in \tau^*(d_1)\}$$

Conversely, τ^* can be expressed in terms of R_τ as follows:

$$\tau^*(x) = \{y \mid (x, y) \in R_\tau\}$$

As Reps et al. show, the representation relation can be pictorially depicted as a bipartite graph on $2(|D|+1)$ vertices. For example, let D be the set $\{a, b, c\}$. Then, the representation relation of the function $\lambda X.(X \cap \{c\}) \cup \{b\}$ can be represented by the following graph [3]:



While Reps et al. abstract this information as a binary relation from D_Λ to D_Λ, it is more convenient for our purpose to abstract it as a function from D_Λ to 2^{D_Λ}. For convenience, we have been treating elements of 2^{D_Λ} as being subsets of D_Λ in the current discussion. We will now switch to treating them as functions from D_Λ to $\{0, 1\}$, and hence, also as functions from D_Λ to $\Re_+$. That is, we can also think of τ^* as being a function from D_Λ to $\Re_+^{D_\Lambda}$.

The next thing to do is to show how to inject a transfer function $\tau \in 2^D \rightarrow_d 2^D$ into a corresponding transfer function $\widetilde{\tau} \in \Re_+^{D_\Lambda} \rightarrow \Re_+^{D_\Lambda}$. We do this as follows[4]:

$$\widetilde{\tau}(\sigma) =_{def} \sum_{d \in D_\Lambda} \sigma(d).\tau^*(d)$$

We will soon see why this definition makes sense, at least for functions that satisfy a simple property. We say that a function $\tau \in 2^D \rightarrow 2^D$ is bi-distributive if it distributes over both set union and set intersection, that is if:

$$\begin{aligned} \tau(X \cup Y) &= \tau(X) \cup \tau(Y) \\ \tau(X \cap Y) &= \tau(X) \cap \tau(Y) \end{aligned}$$

In terms of the representation relation discussed above, a distributive function is bi-distributive iff every vertex in the representation relation has at most one incoming edge. The usual "bit vector" problems, such as reaching definitions, available expressions, and live variables, have bi-distributive transfer functions. Copy constant propagation (eg., see [20]) is an example that is not a "bit vector" problem but *is* a bi-distributive problem. On the other hand, the "possibly-uninitialized variables" problem is *not* bi-distributive: after a statement "$x \;:=\; y + z$", x is possibly-uninitialized if y *or* z is possibly-uninitialized before the statement; thus, the representation relation for the associated transfer function will contain two incoming edges for the vertex corresponding to the fact "x is possibly-uninitialized".

Lemma 2 $\widehat{\tau(\sigma)} = \widetilde{\tau}(\widehat{\sigma})$, *provided* τ *is bi-distributive*

Proof See Appendix. □

The above lemma essentially says that the following diagram commutes (recall that $i(\sigma) = \widehat{\sigma}$):

$$\begin{tikzcd} 2^D \arrow[r, "i"] \arrow[d, "\tau"'] & \Re_+^{D_\Lambda} \arrow[d, "\widetilde{\tau}"] \\ 2^D \arrow[r, "i"] & \Re_+^{D_\Lambda} \end{tikzcd}$$

Remember that our intention is to move the analysis from domain 2^D to domain $\Re_+^{D_\Lambda}$. The commutativity of the above diagram indicates that we are on the right track.

We are not done yet. Now, consider the transfer function $M(e)$ associated with an edge e. We define a new transfer function $\overline{M}(e) : \Re_+^{D_\Lambda} \rightarrow \Re_+^{D_\Lambda}$ by:

$$\overline{M}(e) =_{def} prob(e).\widetilde{M(e)}$$

We can extend the transfer function to paths as usual.

Let us now recap what we have at this stage: we have a domain $\Re_+^{D_\Lambda}$ of values, we have a summary function $+$ defined on $\Re_+^{D_\Lambda}$ and we have associated every edge e of the control-flow graph with a transfer function $\overline{M}(e) \in \Re_+^{D_\Lambda} \rightarrow \Re_+^{D_\Lambda}$. We have essentially a dataflow analysis framework, albeit one in which the summary operator is not idempotent.

The following lemma establishes that performing abstract execution in this new framework is equivalent to performing abstract execution in the original framework and weighting the result by the probability of execution of the path. Thus, we have folded in the probabilities into the framework.

Lemma 3 *Assume that* $M(e)$ *is bi-distributive for every edge* e*. Then, for any path* q *in the graph and any* $\sigma \in 2^D$

$$\overline{M}(q)(\widehat{\sigma}) = prob(q).\widehat{M(q)(\sigma)}$$

Proof See Appendix. □

We will now establish that the sum-over-all-paths quantity computed using this framework yields the values we set out to compute in the first place.

[3] In the context of standard dataflow analysis, the representation relation serves, among other things, to replace the domain 2^D by the domain 2^{D_Λ}. This has the side benefit that the transfer functions can be extended to be truly distributive: that is, to satisfy $f(\phi) = \phi$. See footnote 1.

With 2^D as the domain, we cannot distinguish between the following two situations: (i) execution reaches a program point, but none of the dataflow facts hold true (ii) execution does not reach the program point at all. Both of these are abstractly represented by the empty set. Once we switch to the powerset of D_Λ, however, we *can* make this distinction. The empty set denotes that execution does not reach the program point, while $\{\Lambda\}$ denotes that execution reaches the program point but none of the dataflow facts hold true.

[4] Let us get ahead of ourselves, for the moment, in order to understand this definition. Consider an edge $u \rightarrow v$ with transfer function τ. Assume that the state at u is $\sigma \in \Re_+^{D_\Lambda}$: for every d, $\sigma(d)$ indicates how often d is expected to hold true at u. Assume $\tau^*(d)$ tells us which facts will hold true at v if d holds true at u. To make things simpler for us, let us also assume that for every fact d whether d holds true at v depends only on (some) one fact at u. Can we say how often each fact d is expected to hold true at v? This is the quantity, $\widetilde{\tau}(\sigma)$, that we are trying to define. The definition follows automatically from this reasoning. Also see Lemma 4 and its proof for some more intuition behind the definition.

Definition 3

$$G(u) =_{def} \sum_{q \in Paths(r,u)} \overline{M}(q)(\widehat{c})$$

Theorem 1 *If $M(e)$ is bi-distributive for every edge e then $G(u) = F(u)$.*

Proof
Follows directly from Lemma 3 and Definition 2. □

This concludes our first task: we have shown that the values we want to compute (defined in Definition 1) are given by the sum-over-all-paths solution to a non-idempotent dataflow analysis framework.

4 The Expected Frequencies as a Least Fixed Point

Our next task is to show how the desired sum-over-all-paths quantity can be computed as the fixed point of a collection of equations. Why doesn't this follow directly from Kildall's result? Primarily because, our "summary" operator $+$ is not a meet or join operator (it is not idempotent) and our set of facts is not a lattice. What we will see in this section is that Kildall's theorem generalizes and holds true despite this.

A function $\tau \in \Re_+^{D_\Lambda} \rightarrow \Re_+^{D_\Lambda}$ is said to be a *linear transformation* if it distributes over addition and scalar multiplication:

$$\tau(\alpha.x + \beta.y) = \alpha.\tau(x) + \beta.\tau(y)$$

It is said to be *continuous* if it distributes over infinite sums as well:

$$\tau(\sum_{i \in I} \alpha_i.x_i) = \sum_{i \in I} \alpha_i.\tau(x_i)$$

Lemma 4 *(a) If $\tau : 2^D \rightarrow 2^D$ is distributive, then $\widetilde{\tau} : \Re_+^{D_\Lambda} \rightarrow \Re_+^{D_\Lambda}$ is a continuous linear transformation. (b) If $\tau : \Re_+^{D_\Lambda} \rightarrow \Re_+^{D_\Lambda}$ is a continuous linear transformation, then so is $\beta.\tau$ for $0 \leq \beta \leq 1$.*

Proof
See Appendix. Those familiar with linear algebra may note that $(\Re_+^{D_\Lambda}, +, .)$ is somewhat like a vector space, except that the underlying set D_Λ is a semiring and not a field. D_Λ serves as a "basis" for $\Re_+^{D_\Lambda}$. The definition of $\widetilde{\tau}$ itself is the standard way of extending a function defined on the basis to a linear transformation on the whole space. (The restriction to distributive τ comes about because $\widetilde{\tau}$ is defined only for distributive τ.) □

Definition 4 *Let us denote the following collection of equations by Q.*

$$x_u = \begin{cases} \sum_{v \rightarrow u} \overline{M}(v \rightarrow u)(x_v) & \text{for every } v \in V - \{r\} \\ \widehat{c} & \text{if } v = r \end{cases}$$

Theorem 2 *If $\overline{M}(e)$ is a continuous linear transformation for every edge e, then the sum-over-all-paths values G form a fixed point of the set of equations Q. (That is, the assignment $< x_u := G(u) >$ satisfies the collection of equations Q.)*

Proof See Appendix. □

We now establish a result similar to Kam and Ullman's result[8, 9] relating MOP and MFP. In the following theorem, the ordering on $\Re_+^{D_\Lambda}$ is the ordering induced by the usual arithmetic ordering on $\Re_+$. That is, given $\sigma_1 \in \Re_+^{D_\Lambda}$ and $\sigma_2 \in \Re_+^{D_\Lambda}$, we say that $\sigma_1 \leq \sigma_2$ iff $\forall d \in D_\Lambda.\sigma_1(d) \leq \sigma_2(d)$.

Theorem 3 *If the tranformation function $\overline{M}(e)$ is a linear transformationfor every edge e, then G is less than or equal to any fixed point of Q.*

Proof See Appendix. □

Theorem 4 *If $\overline{M}(e)$ is a continuous linear transformation for every edge e, then G is the least fixed point of the set of equations Q.*

Proof Immediate from Theorems 2 and 3. □

It follows from the above theorem and Lemma 4 that the least fixed point of Q will yield the expected frequencies for finite bi-distributive subset problems.

5 Computing the Least Fixed Point

We now turn to the question of how to compute the desired information. We begin by looking more closely at the set of equations Q. Observe that the domain of the variables in the set of equations Q is $\Re_+^{D_\Lambda}$ — that is, each variable x_u corresponds to a $|D|+1$-dimensional vector. By replacing the vector variable x_u by the set of scalar variables $\{y_{(u,d)} | d \in D_\Lambda\}$ (where $y_{(u,d)}$ corresponds to $x_u(d)$), we can break down each equation in Q into $|D| + 1$ equations.

The structure of the resulting set of equations is better understood by considering the *exploded control-flow graph* introduced by Reps et al. [17]. The exploded graph is obtained by replacing every vertex u in the control-flow graph by $|D| + 1$ vertices $\{(u, d) \mid d \in D_\Lambda\}$ and by replacing every edge $u \rightarrow v$ by the set of edges $\{(u, d_1) \rightarrow (v, d_2) \mid (d_1, d_2) \in R_{M(u \rightarrow v)}\}$. Recall that R_τ denotes the representation relation of the function τ. Thus, the edges in the control-flow graph are replaced by the representation relation of the transfer function associated with the edge. The original dataflow analysis problem reduces to that of reachability in the exploded graph (in the single procedure case).

Every vertex (u, d) in the exploded graph corresponds to our new scalar variable $y_{(u,d)}$. Let us label every edge $(u, d_1) \rightarrow (v, d_2)$ in the exploded graph with the same probability as that of the corresponding edge in the original graph:

$prob((u, d_1) \rightarrow (v, d_2)) =_{def} prob(u \rightarrow v)$. Now Q reduces to a set of equations of the form

$$y_t = \begin{cases} \widehat{c}(d) & \text{if } t = (r, d) \\ \sum_{s \rightarrow t} prob(s \rightarrow t).y_s & \text{otherwise} \end{cases}$$

(Here, the summation is taken over all edges $s \rightarrow t$ in the exploded graph. Also, recall that r is the entry vertex of the original graph.)

Thus, we have a set of *linear* equations with a particularly simple form. In fact, they reveal that just as dataflow analysis reduces to reachability in the exploded graph, the problem we are interested in reduces to a simple algebraic path problem (see [7, 5]) over the exploded graph — namely the algebraic path problem over the closed semiring $(\Re_+, +, ., ^*, 0, 1)$ whose closure operator * is defined by:

$$\alpha^*(= \sum_{i=0}^{\infty} \alpha^i) =_{def} \begin{cases} 1/(1-\alpha) & \text{if } \alpha < 1 \\ \infty & \text{if } \alpha \geq 1 \end{cases}$$

(This algebraic path problem is known as the *Markov chain* problem[5].) Consequently, any of the standard algorithms for solving algebraic path problems can be used here. These algorithms have a worst-case complexity of $O(n^3)$, where n is the number of vertices in the exploded graph (which is $O(|V|.|D|)$).

The algorithms for the algebraic path problem are usually inefficient for dataflow analysis because they fail to utilize the sparsity of the graph and because they end up computing unnecessary information (in the form of the solution to the "all pairs" problem). For our purposes, it would be more efficient to adapt the various elimination-based dataflow analysis algorithms to solve this problem. (See [18] for a survey of elimination algorithms.) These algorithms are often linear or almost-linear in the size of the graph. (The Allen-Cocke interval analysis algorithm [1], for example, is linear for reducible flow graphs whose loop nesting depth is bounded by a constant. Tarjan's algorithm [22] is almost-linear for all reducible flow graphs. These algorithms for work for arbitrary graphs as well, but their complexity can be higher for irreducible graphs.)

We can apply the elimination algorithms to either the original graph or the exploded graph. The exploded graph is larger — the number of edges in the exploded graph is $O(|E|.|D|)$, where $|E|$ is the number of edges in the original graph; this is because every vertex in the representation relation of a bi-distributive function has at most one incoming edge — but the closure and composition of the corresponding dataflow functions can be computed in constant time. The original graph is smaller, but the closure and composition of the corresponding dataflow functions can be more expensive — $O(|D|^3)$ in the worst case. It appears to be difficult to determine which of these two alternatives is better based on the worst-case complexity measure. In general, it appears to be better to apply the elimination algorithm to the exploded graph as it exploits the sparsity of the graph in performing the closure and composition of dataflow functions. However, it does have the drawback that the exploded graph may be irreducible even when the original graph is reducible.

We now address an issue that may come to the mind of some readers. A set of linear equations over real variables usually has (a) a unique solution, or (b) multiple solutions, or (c) no solution. How do these possibilities relate to our problem, which involves a set of linear equations over the domain $\Re_+$? The addition of ∞ (and the restriction to positive reals) eliminates possibility (c) — a solution always exists. When multiple solutions exist, the existence of a least fixed point is also guaranteed (because of the restriction to positive reals).

These possibilities are seen best when we consider how a variable is eliminated from a recursive equation. Consider an equation like

$$y_u = \alpha_u.y_u + z$$

where z is an expression not containing y_u. If $\alpha_u < 1$, then this equation reduces to the non-recursive

$$y_u = (1/(1-\alpha_u)).z$$

If $\alpha_u \geq 1$ and z is non-zero, then the equation has no "finite" solution. The unique solution, within our domain, is $y_u = \infty$. If $\alpha_u = 1$ and $z = 0$, then there are infinitely many solutions for y_u, but $y_u = 0$ is the least fixed point (which is what we want to compute). If $\alpha_u > 1$ and $z = 0$, then the solutions are $y_u = \infty$ and $y_u = 0$, with the latter being the least fixed point.

All of the above possibilities are captured precisely by the following transformed non-recursive equation:

$$y_u = \alpha_u^*.z$$

where the closure operator * is defined as previously. (Recall that $\infty.0 = 0$.)

Note that the closure operator * is a constant time operation. Use of the closure operator and elimination techniques imply that we do not have to worry about the "convergence" of the algorithm.

6 Handling Interprocedural Analysis

Procedures add a new twist to the meet-over-all-paths formulation of dataflow analysis problems. A program with multiple procedures is represented by a supergraph that contains a control-flow graph G_p for every procedure p. Each control-flow graph G_p has its own entry vertex, denoted by r_p, and its own exit vertex, denoted by $exit_p$. The entry vertex of a unique procedure, denoted $main$, is considered to be the entry vertex of the whole program. A procedure call-site s is represented by two vertices, the call vertex $call_s$ and the

[5] One minor difference is that in Markov chains it is assumed that the sum of the probabilities over all edges going out of a vertex is 1. While this is true for the original control-flow graph, it need not be true in the exploded graph.

return-site vertex $return_s$ and two edges $call_s \rightarrow r_p$ (the call edge) and $exit_p \rightarrow return_s$ (the return edge), where p is the procedure called.

A precise interprocedural dataflow analysis algorithm computes the "meet-over-all-valid-paths" solution, as opposed to the "meet-over-all-paths" solution [21, 11]. If one considers the corresponding call edge $call_s \rightarrow r_p$ and return edge $exit_p \rightarrow return_s$ to be matching open and close parentheses, then a path is said to be a *same level valid path* if the sequence of call and return edges in the path forms a well-balanced sequence of parentheses. A path is said to be a *valid path* if it can be expressed as the concatenation of zero or more same level valid paths and call edges. We will denote by $\mathit{VP}(u, v)$ and $\mathit{SLVP}(u, v)$ the set of all valid paths and same level valid paths, respectively, from u to v.

We generalize Definition 3, in the interprocedural case, to take the sum over all valid paths. (All call and return edges are labelled with a probability of 1.) We now show how we can compute this quantity by adapting existing interprocedural and intraprocedural analysis techniques. The algorithm consists of the following phases.

Phase 1: Compute summary information for each procedure. We define the *summary* of any set of paths P to be the quantity $\sum_{p \in P}(prob(p).\overline{M}(p))$, which we denote by $\overline{M}(P)$. The quantity $\overline{M}(P)$ is itself a linear transformation on $\Re_+^{D^\Lambda}$, which can be represented by a $(|D|+1) \times (|D|+1)$ matrix of real values or by a labelled bipartite graph from $|D|+1$ vertices to $|D|+1$ vertices, just like a labelled representation relation.

The first phase of the algorithm is to compute summary information for each procedure p that summarizes the set of all same level valid paths from r_p to $exit_p$ — *i.e.*, the quantity $\overline{M}(\mathit{SLVP}(r_p, exit_p))$. In the absence of recursion, this information can be computed bottom-up over the call graph using the intraprocedural algorithm for each procedure. In the presence of recursion, however, this information has to be computed iteratively, as below.

Introduce a symbolic variable X_p to denote the summary information of each procedure p. (As explained earlier, X_p represents $(|D|+1) \times (|D|+1)$ scalar variables.) Replace all interprocedural edges by summary edges: for every call-site s to a procedure q, introduce an edge between $call_s$ and $return_s$ whose transfer function is X_q. Now, analyze each procedure separately.

The analysis of each control-flow graph, however, cannot be done as before (in the intraprocedural case), since the edges of the representation relation of the summary edges are labelled with symbolic variables instead of real values. Instead, we perform the analysis using an adaptation of Tarjan's generalized path algorithm [22]. This algorithm performs the elimination symbolically and computes a (symbolic) regular expression (*i.e.*, an expression containing the operators +, ., and *) for each vertex u in the procedure p. This quantity, in our case, essentially expresses $\overline{M}(\mathit{SLVP}(r_p, u))$ as a symbolic expression over the set of variables $\{X_q \mid p \text{ calls } q\}$.

Consider, in particular, the value computed for the exit vertex of all procedures. This expresses the summary function of each procedure as an expression over the summary function of the procedures it calls. We solve this collection of equations using iteration (until we numerically converge to a fixed point). Unlike in the intraprocedural case, we may now have non-linear equations, such as $x = x^2/2 + x/4 + 1/4$, which is why we need to use iterative techniques, instead of elimination, to solve the equations.

There is, however, one problem using iterative techniques here: the iteration may not converge if there is "infinite recursion" in the call graph (*i.e.*, if the expected execution frequency of some procedure is ∞ in the fixed point). One possible way of tackling this problem is to first compute just the expected execution frequency of each procedure, which can be done using just elimination techniques [23, 24]. If none of these values are ∞ we may proceed with the iterative techniques to compute the frequency of dataflow facts. If any of these values is ∞, we can heuristically alter the "probability" labelling, as in [23], so that these values are finite.

Once the summary values have been computed, every $\overline{M}(\mathit{SLVP}(r_p, u))$ (where u is a vertex in procedure p) can now be computed by evaluating the corresponding symbolic expression.

Phase 2: Compute the solution for the entry vertex of all procedures: Construct the call graph of the program, representing each *call site* by an edge from the calling procedure to the called procedure. Consider a call-site s in a procedure p. Associate the edge corresponding to this call-site with the transfer function $\overline{M}(\mathit{SLVP}(r_p, call_s))$ — this quantity is readily available from the results of the analysis of the first phase. Apply the intra-procedural algorithm to the call-graph — this gives us the meet-over-all-valid-paths solution for the entry vertex of each procedure.

Phase 3: Compute the rest of the solution: The meet-over-all-valid-paths solution for a vertex u in a procedure p can be obtained by applying $\overline{M}(\mathit{SLVP}(r_p, u))$ to the solution for r_p.

7 Conclusion

This work was originally motivated by *procedure cloning* — an optimization that generates specializations of procedures for specific subtypes of the types of their formal parameters [6]. More efficient code can be generated for the specialized versions than for the unspecialized version because the types of the arguments are more precisely known. However, it is necessary to be selective about which specializations are generated to avoid code explosion. This requires knowledge about which procedures are executed often and about how frequently arguments of different types are passed to the various procedures. Such information is typically gathered at run-time using tracing tools that record the relevant information.

This paper explores the possibility of computing such information statically *once every branch out of a conditional* *node is labelled with a value that indicates the probability that that branch is taken*. The probability labelling itself may be computed using simple heuristics, profiling tools, or static analysis. [23, 24, 14, 3]. Even when the labelling is computed using profiling tools, it is certainly more convenient to have to gather only this simple data at run-time, rather than having to gather, at run-time, specialized information that varies from one analysis problem to another. Profile data, unlike the problem-specific dataflow information, can be gathered using existing "off-the-shelf" execution profilers.

This paper is a first step towards addressing this problem. It presents a theoretical foundation, as well as algorithms, for computing the frequency of dataflow facts.

More work, however, remains to be done. Consider the type analysis problem that arises in the above-mentioned cloning optimization. There do exist simple, but effective, type analysis algorithms (for C++; see [2]) that fall into the FBS framework. Other, more powerful, type analysis algorithms [12, 13, 4], however, are not even distributive. The results in this paper do not apply to such algorithms. It remains to be seen what can be done about such algorithms. (At the least, there is hope that for finite monotonic dataflow analysis problems we can formulate the appropriate equations whose least fixed point is an *upper bound* on the frequency of dataflow facts, extending the corresponding result for monotonic dataflow analysis.)

A second issue concerns the fact that the problem requires solving equations over real-valued variables, unlike traditional dataflow analysis problems which deal with "discrete" lattices. Consequently, we need to deal with issues such as "precision" and "error magnification". (For instance, how sensitive is the solution to the initial probability assignment? Can a small change in the probability assignment cause a big change in the final solution?) Hence, in designing algorithms to solve this problem, it seems appropriate to go beyond traditional dataflow elimination algorithms and look at elimination and iteration schemes used in the area of numerical analysis to deal with such issues. A related question that needs to be addressed concerns the convergence and complexity of the iterative methods used in the interprocedural case. (The intraprocedural problem can be solved using elimination techniques, where convergence is not a problem.)

A third issue that deserves consideration is that we assume that the probability of the program execution following a particular branch is independent of the execution history. It remains to be seen how much this approximation can affect the solution.

The problem of computing the expected frequency of execution of the various statements in a program, given a control-flow graph whose edges are labelled with a probability, has been studied previously [16, 23, 24]. A couple of recent papers have utilized estimates of the "probability" of specific "facts" for program optimization. Proebsting and Fischer [15] outline a register allocation algorithm that utilizes estimates of the probability of a register-allocated variable remaining in the register at a subsequent use of the variable. Patterson [14] computes the probability of various variables having various values and utilizes this information for branch prediction, which has applications to several optimizations such as instruction scheduling. These papers, however, present no results relating the quantities computed to any formally defined quantity (such as a sum-over-all-path quantity). Nor do they explore the problem in the generalized setting that we do.

The exploded control-flow graph introduced by Reps et al. [17] provides a natural and easy way to understand our problem reduction and algorithm. There are, in fact, several structural similarities between our work and an extension of the Reps et al. work outlined by Sagiv et al. [20, 19]. Sagiv et al. study a class of dataflow problems they call the *Interprocedural Distributive Environment* (IDE) problems. The similarities arise from the fact that while Sagiv et al. show that the IDE problems reduce to a "simpler" dataflow problem (*i.e.*, one involving a simpler lattice) over the exploded graph, we show that the dataflow frequency problem (for FBS problems) reduces to a simple Markov chain problem over the exploded graph. The primary difference between the final reduced form of the problems in the two cases is that the summary operator is idempotent in one case but not the other.

Acknowledgements

The author thanks the anonymous referees for their helpful comments.

References

[1] F.E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. of the ACM*, 19(3):137–147, March 1976.

[2] D. F. Bacon, M. Wegman, and K. Zadeck. Rapid type analysis for C++. Technical Report RC number pending, IBM T.J. Watson Research Center, 1996.

[3] T. Ball and J.R. Larus. Optimally profiling and tracing programs. In *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages*, pages 59–70, 1992.

[4] Paul R. Carini, Michael Hind, and Harini Srinivasan. Flow-sensitive type analysis for C++. Technical Report RC 20267, IBM T.J. Watson Research Center, 1995.

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[6] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming*

Language Design and Implementation, pages 93–102, 1995.

[7] M. Gondran and M. Minoux. *Graphs and Algorithms*. John Wiley and Sons, New York, 1984.

[8] J.B. Kam and J.D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23:158–171, 1976.

[9] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[10] G. Kildall. A unified approach to global program optimization. In *Conference Record of the First ACM Symposium on Principles of Programming Languages*, pages 194–206, New York, NY, 1973. ACM.

[11] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In U. Kastens and P. Pfahler, editors, *Proceedings of the Fourth International Conference on Compiler Construction, Lecture Notes in Computer Science Vol. 641*, pages 125–140. Springer Verlag, 1992.

[12] Hemant D. Pande and Barbara G. Ryder. Static type determination for C++. In *Proceedings of the Sixth Usenix C++ Technical Conference*, pages 85–97, April 1994.

[13] Hemant D. Pande and Barbara G. Ryder. Static type determination and aliasing for C++. Technical Report LCSR-TR-250, Department of Computer Science, Rutgers University, July 1995.

[14] J.R.C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 67–78, 1995.

[15] T. Proebsting and C. Fischer. Probabilistic register allocation. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 300–310, 1992.

[16] C.V. Ramamoorthy. Discrete markov analysis of computer programs. In *Proceedings of the ACM 20th National Conference*, pages 386–391, 1965.

[17] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[18] B. Ryder and M. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3), September 1986.

[19] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. To appear in *Theoretical Computer Science*. (Also Technical Report CS-TR-95-1284, Computer Sciences Department, University of Wisconsin-Madison, 1995).

[20] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Proceedings of FASE '95: Colloquium on Formal Approaches in Software Engineering*, pages 651–665, 1995.

[21] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[22] R.E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28:594–614, 1981.

[23] T.A. Wagner, V. Maverick, S.L. Graham, and M.A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 85–96, 1994.

[24] Y. Wu and J.R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 1–11, 1994.

Appendix

Proof of Lemma 1

$$
\begin{aligned}
F(u)(\Lambda) &= \Big(\sum_{q \in Paths(r,u)} prob(q).\ \widehat{state}(q) \Big)\ (\Lambda) \\
&= \sum_{q \in Paths(r,u)} \big(prob(q).\ (\widehat{state}(q)\ (\Lambda)) \big) \\
&= \sum_{q \in Paths(r,u)} prob(q) \\
&\qquad \text{(using equation 1)} \\
&= E(u)
\end{aligned}
$$

Similarly, for any $d \in D$,

$$
\begin{aligned}
F(u)(d) &= \Big(\sum_{q \in Paths(r,u)} prob(q).\widehat{state}(q) \Big)\ (d) \\
&= \sum_{q \in Paths(r,u)} \big(prob(q).\ (\widehat{state}(q)(d)) \big) \\
&= \sum_{q \in Paths(r,u)} \big(prob(q).\ (state(q)(d)) \big) \\
&\qquad \text{(using equation 1)}
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{\substack{q \in Paths(r,u) \\ state(q)(d)=1}} prob(q) \\
&\qquad \text{(since } state(q)(d) \in \{0,1\}) \\
&= E(u,d)
\end{aligned}
$$

□

Lemma 5 *Let $\tau \in 2^D \to 2^D$ be bi-distributive and $x \in D$. Then (a) if $x \in \tau(\sigma)$ then there exists an unique $d \in \widehat{\sigma}$ such that $x \in \tau^*(d)$ (b) if $x \notin \tau(\sigma)$ then for all $d \in \widehat{\sigma}$ $x \notin \tau^*(d)$.*

Proof

(a) We have $x \in \tau(\sigma)$. First consider the case where $x \in \tau(\phi)$. Here, we have $x \in \tau^*(\Lambda)$ and $x \notin \tau^*(d)$ for any $d \in D$. (This follows directly from the definition of τ^*.)

Now assume that $x \notin \tau(\phi)$. Since τ is distributive, $\tau(\sigma) = \tau(\cup_{d \in \sigma}\{d\}) = \cup_{d \in \sigma}\tau\{d\}$. Hence, there must be some $d \in \sigma$ such that $x \in \tau\{d\}$. This d has to be unique: if $x \in \tau\{d_1\}$ and $x \in \tau\{d_2\}$, where $d_1 \neq d_2$ then we have $x \in \tau(\{d_1\}) \cap \tau(\{d_2\}) = \tau(\{d_1\} \cap \{d_2\})$ (since τ is bi-distributive) $= \tau(\phi)$. This contradicts our assumption that $x \notin \tau(\phi)$. Hence, it follows that there is an unique $d \in \widehat{\sigma}$ such that $x \in \tau^*(d)$.

(b) This follows directly from the monotonicity of τ: if $x \notin \tau(\sigma)$ then for all $d \in \sigma$ $x \notin \tau(\{d\})$ and $x \notin \tau(\phi)$. The result follows. □

Proof of Lemma 2

We need to show that $\widehat{\tau(\sigma)}(x) = \widetilde{\tau}(\widehat{\sigma})(x)$, for all $x \in D_\Lambda$.

First consider any $x \in \tau(\sigma)$ — or, more precisely, any $x \in D$ such that $\tau(\sigma)(x) = 1$. Lemma 5 implies that there exists an unique $d \in \widehat{\sigma}$ such that $x \in \tau^*(d)$ — or, more precisely, there exists an unique $d \in D_\Lambda$ such that $\widehat{\sigma}(d) = 1$ and $\tau^*(d)(x) = 1$. Consequently, $\widehat{\sigma}(d).\tau^*(d)(x) = 0$ for all other $d \in D_\Lambda$. Thus, we have

$$
\begin{aligned}
\widetilde{\tau}(\widehat{\sigma})(x) &= \Big(\sum_{d \in D_\Lambda} \widehat{\sigma}(d).\tau^*(d)\Big)\ (x) \\
&= \sum_{d \in D_\Lambda} (\ \widehat{\sigma}(d).\ \tau^*(d)(x)\ ) \\
&= 1 \quad \text{(since there is a unique } d \text{ such that } \widehat{\sigma}(d).\ \tau^*(d)(x) = 1) \\
&= \widehat{\tau(\sigma)}\ (x)
\end{aligned}
$$

Now consider any $x \notin \tau(\sigma)$ — or, more precisely, any $x \in D$ such that $\tau(\sigma)(x) = 0$. The previous lemma implies that there for all $d \in \widehat{\sigma}$ $x \notin \tau^*(d)$ — or, more precisely, for all $d \in D_\Lambda$ if $\widehat{\sigma}(d) = 1$ then $\tau^*(d)(x) = 0$. In other words, $\widehat{\sigma}(d).\tau^*(d)(x) = 0$ for all $d \in D_\Lambda$. Consequently, we have

$$
\begin{aligned}
\widetilde{\tau}(\widehat{\sigma})(x) &= \Big(\sum_{d \in D_\Lambda} \widehat{\sigma}(d).\tau^*(d)\Big)\ (x) \\
&= \sum_{d \in D_\Lambda} (\ \widehat{\sigma}(d).\ \tau^*(d)(x)\ ) \\
&= 0 \\
&= \widehat{\tau(\sigma)}\ (x)
\end{aligned}
$$

Now consider $x = \Lambda$.

$$
\begin{aligned}
\widetilde{\tau}(\widehat{\sigma})(\Lambda) &= \Big(\sum_{d \in D_\Lambda} \widehat{\sigma}(d).\tau^*(d)\Big) && (\Lambda) \\
&= \Big(\widehat{\sigma}(\Lambda).\tau^*(\Lambda) + \sum_{d \in D} \widehat{\sigma}(d).\tau^*(d)\Big) && (\Lambda) \\
&= \Big(\widehat{\tau(\phi)} + \sum_{d \in D} \sigma(d).(\tau(\{d\}) - \tau(\phi))\Big) && (\Lambda) \\
&= \widehat{\tau(\phi)}(\Lambda) + \sum_{d \in D} \sigma(d).(\tau(\{d\}) - \tau(\phi)) && (\Lambda) \\
&= 1 + \sum_{d \in D} \sigma(d).0 \\
&= 1 \\
&= \widehat{\tau(\sigma)}(\Lambda)
\end{aligned}
$$

□

Proof of Lemma 3

This follows by induction on the path length. The lemma follows trivially for the empty path (whose transfer function is the identity function in both frameworks). Assume that the result holds for a specific path q. We will show that it holds true for the path $q.e$ (consisting of the path q followed by the edge e).

$$
\begin{aligned}
\overline{M}(q.e)\ (\widehat{\sigma}) &= \overline{M}(e)\ \ (\ \overline{M}(q)(\widehat{\sigma})\ ) \\
&= \overline{M}(e)\ \ (\ prob(q).\widehat{\widetilde{M(q)(\sigma)}}\ ) \\
&\qquad \text{(using the induction hypothesis)} \\
&= (prob(e).\widetilde{M(e)})\ \ (prob(q).\widehat{\widetilde{M(q)(\sigma)}}) \\
&\qquad \text{(using the definition of } \overline{M}(e)) \\
&= prob(e).prob(q).(\widetilde{M(e)}(\widehat{\widetilde{M(q)(\sigma)}})) \\
&\qquad \text{(since } \widetilde{M(e)} \text{ is a linear transformation)} \\
&= prob(q.e).\widehat{\widetilde{(M(e)(M(q)(\sigma)))}} \\
&\qquad \text{(using Lemma 2)} \\
&= prob(q.e).\widehat{\widetilde{(M(q.e)(\sigma))}}
\end{aligned}
$$

□

Proof of Lemma 4

(a) Assume that $\tau : 2^D \to 2^D$ is distributive.

$$
\widetilde{\tau}\Big(\sum_{i \in I} \alpha_i.x_i\Big) = \sum_{d \in D_\Lambda} \Big(\sum_{i \in I} \alpha_i.x_i\Big)(d).\tau^*(d)
$$

$$
\begin{aligned}
&= \sum_{d\in D_\Lambda}\sum_{i\in I}(\ \alpha_i.x_i(d).\tau^*(d)\ )\\
&= \sum_{i\in I}\sum_{d\in D_\Lambda}(\ \alpha_i.x_i(d).\tau^*(d)\ )\\
&= \sum_{i\in I}\alpha_i.(\sum_{d\in D_\Lambda}(\ x_i(d).\tau^*(d)\ )\\
&= \sum_{i\in I}\alpha_i.\widetilde{\tau}(x_i)
\end{aligned}
$$

(b) Assume that $\tau : \Re_+^{D_\Lambda} \rightarrow \Re_+^{D_\Lambda}$ is a continuous, linear transformation.

$$
\begin{aligned}
(\beta.\tau)(\sum_{i\in I}\alpha_i.x_i) &= \beta.(\tau(\sum_{i\in I}\alpha_i.x_i))\\
&= \beta.(\sum_{i\in I}\alpha_i.\tau(x_i))\\
&= \sum_{i\in I}(\alpha_i.(\beta.\tau)(x_i))
\end{aligned}
$$

□

Proof of Theorem 2

The result trivially holds true for the entry vertex r, since the empty path is the only path from r to r. (Recall that we have assumed that the entry vertex has no predecessors.) Consider any vertex u other than the entry vertex.

$$
\begin{aligned}
G(u) &= \sum_{q\in Paths(r,u)} \overline{M}(q)(\widehat{c})\\
&= \sum_{v\rightarrow u}\sum_{q'\in Paths(r,v)} \overline{M}(v\rightarrow u)(\overline{M}(q')(\widehat{c}))\\
&= \sum_{v\rightarrow u}\overline{M}(v\rightarrow u)(\sum_{q'\in Paths(r,v)}\overline{M}(q')(\widehat{c}))\\
&\quad \text{(since } \overline{M}(v\rightarrow u) \text{ is linear)}\\
&= \sum_{v\rightarrow u}\overline{M}(v\rightarrow u)(G(v))
\end{aligned}
$$

□

Proof of Theorem 3

Let $(s_u \mid u$ is a vertex$)$ be a fixed point of the set of equations Q. Let T be any finite set of paths from the entry vertex r to a vertex u. We now show that $\sum_{q\in T}\overline{M}(q)(\widehat{c}) \leq s_u$. We do this by induction over the sum of the lengths of the paths in T.

There are two base cases: (i) T is empty: in this case, $\sum_{q\in T}\overline{M}(q)(\widehat{c})$ is 0 (i.e., $\lambda d.0$) and the result holds (since the fixed point is an element of $\Re_+^{D_\Lambda}$). (ii) T is a singleton consisting of the empty path: In this case, u must be the entry vertex r. Here, $\sum_{q\in T}\overline{M}(q)(\widehat{c})$ is $\widehat{c}$, and the result holds true.

Now for the induction step. For every predecessor v of u, let T_v denote the set of paths in T that end with edge $v \rightarrow u$.

Note that $T = \cup_{v\rightarrow u}\cup_{p\in T_v}\{p.(v\rightarrow u)\}$. We have

$$
\begin{aligned}
\sum_{q\in T}\overline{M}(q)(\widehat{c}) &= \sum_{q\in T}\overline{M}(q)(\widehat{c})\\
&= \sum_{v\rightarrow u}(\sum_{p\in T_v}\overline{M}(p.(v\rightarrow u))(\widehat{c})\\
&= \sum_{v\rightarrow u}(\sum_{p\in T_v}\overline{M}(v\rightarrow u)(\overline{M}(p)(\widehat{c})\\
&= \sum_{v\rightarrow u}\overline{M}(v\rightarrow u)(\sum_{p\in T_v}(\overline{M}(p)(\widehat{c})))\\
&\quad \text{(since } \overline{M}(v\rightarrow u) \text{ is linear)}\\
&\leq \sum_{v\rightarrow u}\overline{M}(v\rightarrow u)s_v\\
&\quad \text{(using the inductive hypothesis)}\\
&\leq s_u\\
&\quad \text{(since } (s_u \mid u \text{ is a vertex) is a fixed point)}
\end{aligned}
$$

The theorem follows since the sum over an infinite set X is simply the limit of the sums over all finite subsets of X. □