A graph model of data and workflow provenance

Umut Acar
Max-Planck Institute for Software Systems

Peter Buneman
University of Edinburgh

James Cheney
University of Edinburgh

Jan Van den Bussche
Hasselt University

Natalia Kwasnikowska
Hasselt University

Stijn Vansummeren
Université Libre de Bruxelles

Abstract

Provenance has been studied extensively in both database and workflow management systems, so far with little convergence of definitions or models. Provenance in databases has generally been defined for relational or complex object data, by propagating fine-grained annotations or algebraic expressions from the input to the output. This kind of provenance has been found useful in other areas of computer science: annotation databases, probabilistic databases, schema and data integration, etc. In contrast, workflow provenance aims to capture a complete description of evaluation – or enactment – of a workflow, and this is crucial to verification in scientific computation. Workflows and their provenance are often presented using graphical notation, making them easy to visualize but complicating the formal semantics that relates their run-time behavior with their provenance records. We bridge this gap by extending a previously-developed dataflow language which supports both database-style querying and workflow-style batch processing steps to produce a workflow-style provenance graph that can be explicitly queried. We define and describe the model through examples, present queries that extract other forms of provenance, and give an executable definition of the graph semantics of dataflow expressions.

1 Introduction

A number of standard database provenance models tailored to relational, complex-object or XML query languages have emerged. These models include lineage [9], where-provenance [3, 2], why-provenance [3, 4], and more recent innovations such as dependency-provenance [7], how-provenance [13, 11], and provenance traces [6]. These models have been presented in a number of different ways and founded on several different motivations. Recently, further study has revealed that these models share a great deal of structure once they are defined in a common language and data model [8, 6].

Provenance models have also been developed for a variety of workflow systems, such as Chimera [10], Taverna [20], Kepler [1], Karma [24], and ZOOM [26]; also, many other systems such as PASS and PASOA employ similar ideas [14, 23]. These systems model and record provenance as a directed acyclic graph that, informally, describes the macroscopic computation steps (e.g., whole program executions) performed in constructing intermediate and final results. Recently, the Open Provenance Model (OPM) [22] has been developed as a consensus exchange format for representing provenance graphs.

Workflow systems employ a much wider variety of programming constructs than databases, including concurrency, procedures, service calls, and queries to external databases. However, these systems are seldom accompanied by formal specifications of their intended semantics, with or without provenance. As a result, it can be hard to understand provenance information produced by a workflow system, since the meaning intended by the implementer may not match the expectations of the user. This is a particularly vexing problem because some users and implementers might not even be aware of the possibility of misinterpretation, leading to further confusion.

The scarcity of clear specifications of the semantics and provenance behavior of workflow systems makes it difficult to integrate database and workflow provenance or compare provenance graphs generated by different systems. Therefore, we believe that it is essential to study the semantics of workflow provenance models and relate them to existing models of database provenance.

To compare and unify these different techniques, we need to define a common provenance model. Database provenance models can be visualized as graphs. Where-provenance, lineage, and dependency-provenance can be visualized as bipartite graphs linking parts of the output with parts of the input. How-provenance and why-

provenance are more complex, but can also be visualized as directed acyclic graphs linking parts of the output to parts of the input, where nodes are labeled with symbolic algebraic operations. Graphs provide a natural common formalism for workflow and database provenance.

We also need a common language that can express both database queries and workflows. In this paper, we use a core calculus for *dataflows*, called DFL, based on the Nested Relational Calculus (NRC). DFL has been previously introduced by Hidders et al. [16, 17] and we also build upon some prior work on provenance in this setting [19]. We develop a graphical model of provenance for both database queries and simple workflows in a uniform way. This should provide a foundation for studying more complex workflow language features such as nondeterminism, concurrency and while-loops.

The structure of the rest of this paper is as follows. In Section 2 we review the dataflow calculus DFL. In Section 3 we describe the structure of provenance graphs and give examples showing how typical dataflows are translated to graphs. In Section 4 we describe the provenance semantics of DFL programs, and give an executable, yet still high-level implementation in Haskell. Section 5 discusses how to express queries over the graphs, particularly inspired by where- and and why-provenance in databases, and outlines some future directions. Section 6 discusses related work and Section 7 concludes.

Note. Our graphical model is fundamentally very similar to the trace model developed in prior, unpublished work with Acar and Ahmed [6]. However, we make a different contribution: namely, we feel our graph-theoretic presentation is more widely accessible than the syntactic traces and operational semantics rules employed to simplify proofs of their main results in [6].

2 Background

The dataflow language DFL [17] is an extension of the Nested Relational Calculus (NRC) that includes atomic values and functions. As we can only briefly introduce DFL here due to space reasons, we encourage readers unfamiliar with DFL and NRC to consult [5, 17] for more details. In brief, the syntax of DFL is as follows:

$$
\begin{array}{rcl}
e, e' & ::= & x \mid \mathsf{let}\ x = e\ \mathsf{in}\ e' \mid c \mid f(e_1, \ldots, e_n) \\
& \mid & \pi_A(e) \mid \langle A_1{:}e_1, \ldots, A_n{:}e_n \rangle \mid \mathsf{empty?}(e) \\
& \mid & \mathsf{True} \mid \mathsf{False} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
& \mid & \emptyset \mid \{e\} \mid e_1 \cup e_2 \mid \{e' \mid x \in e\} \mid \bigcup e
\end{array}
$$

Here, c denotes a constant atomic data value, drawn from a set D, and f denotes a function. Atomic data values may be values of base types such as integers or booleans or strings, but they may also be more complicated objects such as images or data files. Functions include primitive operations on basic data types, such as integer addition and equality. Furthermore, functions can also represent large computational steps such as external program or service calls: for example, to model the first Provenance Challenge workflow we might use base types such as Image, Header, or WarpFile and function symbols such as $\mathsf{align_warp} : \mathsf{Image} \times \mathsf{Header} \times \mathsf{Image} \times \mathsf{Header} \to \mathsf{WarpFile}$ or $\mathsf{reslice} : \mathsf{WarpFile} \to \mathsf{Image} \times \mathsf{Header}$ to represent the macroscopic computation steps.

The remaining syntactic constructs above are standard components of the Nested Relational Calculus: we include record and field projection operations, booleans and conditionals, and set operations. We employ the syntax $\{e'(x) \mid x \in e\}$ for the "for-loop" or set comprehension operation which evaluates e to a set $\{v_1, \ldots, v_n\}$ and returns the set of values $\{e'(v_1), \ldots, e'(v_n)\}$ obtained by evaluating e' with x bound to each v_i. The expression $\bigcup e$ flattens a nested collection. The expression $\mathsf{empty?}(e)$ tests whether collection e is empty.

We will use ordered-pair syntax (e_1, e_2) to abbreviate $\langle \mathsf{fst} : e_1, \mathsf{snd} : e_2 \rangle$, and write $\mathsf{fst}(e)$ or $\mathsf{snd}(e)$ instead of $\pi_{\mathsf{fst}}(e)$ or $\pi_{\mathsf{snd}}(e)$, respectively, for the first and second projections of an ordered pair. We also assume a fixed, finite set of attribute names Attr.

DFL and NRC are statically typed languages with an arbitrary but fixed collection of atomic types, and an arbitrary but fixed signature that assigns types to the constants and function symbols [5]. The static typing discipline ensures that expressions are always well-defined on input values of the correct type. For ease of presentation in what follows, we will ignore typing issues and silently restrict attention to expressions and evaluations that are well-defined in the conventional sense [5]. So whenever we apply, for example, $e_1 \cup e_2$, e_1 and e_2 are assumed to correctly evaluate to sets.

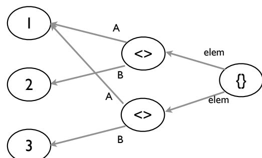
3 Value, evaluation and provenance graphs

DFL expressions are normally evaluated over *complex values*, which are nested combinations of atomic data values d, tuples of complex values $\langle A_1 : v_1, \ldots, A_n : v_n \rangle$, and sets of complex values $\{v_1, \ldots, v_n\}$. As we show in Section 3.1, we can easily represent complex values as trees or (with sharing) as directed acyclic graphs. Using such *value graphs*, we are going to represent the evaluation of a DFL expression by means of a *provenance graph* in Section 3.3. A provenance graph is a two-sorted graph, consisting of a value graph and an *evaluation graph* (introduced in Section 3.2), that documents the evaluation of a program. Moreover, there is a connection between the evaluation graph and the value graph in that each evaluation node is linked to a value node.

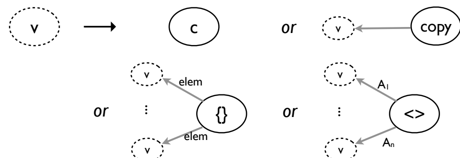
3.1 Value graphs

A *value graph* G is a directed acyclic graph (V, E) with labels on the nodes and edges. The nodes are labeled using the alphabet $\{\{\}, \langle\rangle, \mathsf{copy}\} \cup D$. The edges are optionally labeled using the alphabet $\{\mathsf{elem}\} \cup \mathsf{Attr}$. We use the formula $\mathsf{lab}_l(n)$ to indicate that n has label l in G and $n \xrightarrow{l} n'$ to indicate that there is an edge (n, n') with label l in G.

To illustrate, the following graph represents the value $\{\langle A:1, B:2\rangle, \langle A:1, B:3\rangle\}$:



We restrict our attention in what follows to *legal* value graphs that can be constructed using the following rules:



The meaning of these patterns is that a value graph v can be constructed from another valid graph by adding new nodes and edges (shown using solid lines) linked to some existing nodes (shown using dotted lines). Sharing among the nodes of the value graph is allowed. Also, the empty graph is valid and the union of two disjoint value graphs is valid.

A tree-shaped value graph is called a *value tree*. Clearly, one can canonically represent any complex value by the root node of a value tree. Moreover, any value graph can be converted to a value tree by duplicating shared nodes and by merging copy nodes with their targets. In any value graph, any node from which the unraveling yields this canonical value tree, is said to also represent the same complex value.

In what follows, we often say "n is a copy of n' (in G)" as shorthand meaning that n is a copy node and its (sole, unlabeled) outgoing edge is n'.

3.2 Evaluation graphs

An *evaluation graph* $G = (V, E)$ is a labeled directed acyclic graph with node labels drawn from the set

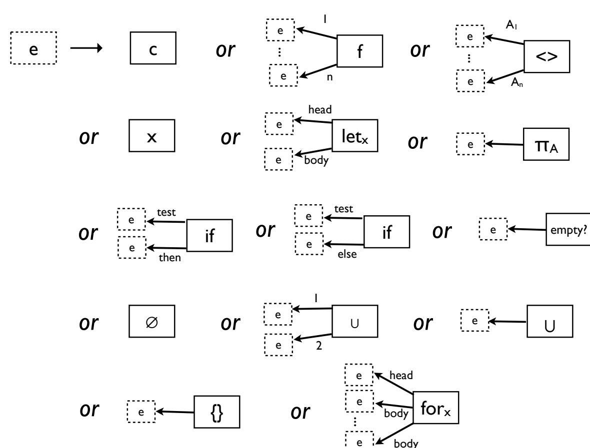
$$\{x, c, f, \langle\rangle, \pi_A, \mathsf{let}_x, \mathsf{if}, \emptyset, \{\}, \cup, \bigcup, \mathsf{for}_x\}$$

and optional edge labels drawn from the set

$$\{A, \mathsf{head}, \mathsf{body}, \mathsf{test}, \mathsf{then}, \mathsf{else}, 1, 2, \ldots\}$$

As with value graphs, we write $lab_l(m)$ to indicate that node x has label l and $m \xrightarrow{l} m'$ to indicate that nodes m and m' are linked by an edge labeled l.

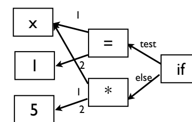
A valid evaluation graph is one that can be constructed using the following rules:



where, again, the meaning of each pattern is that we can extend the graph by adding new nodes and edges (shown using solid lines) by linking to existing nodes (shown using dotted lines). The existing nodes need not be disjoint, so sharing can occur in evaluation graphs. Also, the empty graph is valid and the union of two disjoint graphs is valid.

Finally, we introduce the following terminology: A node labeled $\mathsf{let}\ x$ or $\mathsf{for}\ x$ is said to *bind* x. We say that a variable node e_x labeled by x is in the scope of a node e that binds x, if there is a path from e to e_x that does not pass through another node that binds x. We require each variable node to be in the scope of at most one binding node.

For example, the following is a valid evaluation graph:



Essentially, this graph says that a value was obtained by doing a conditional test $x = 1$ which failed, and then evaluating the else-branch to return $x * 5$. Note that there is no information about the actual value of x or the result of the computation, although we can infer that $x \neq 1$ holds.

3.3 Provenance Graphs

A *provenance graph* $G = (V, E, val)$ is a directed acyclic graph with nodes and edges labeled with either

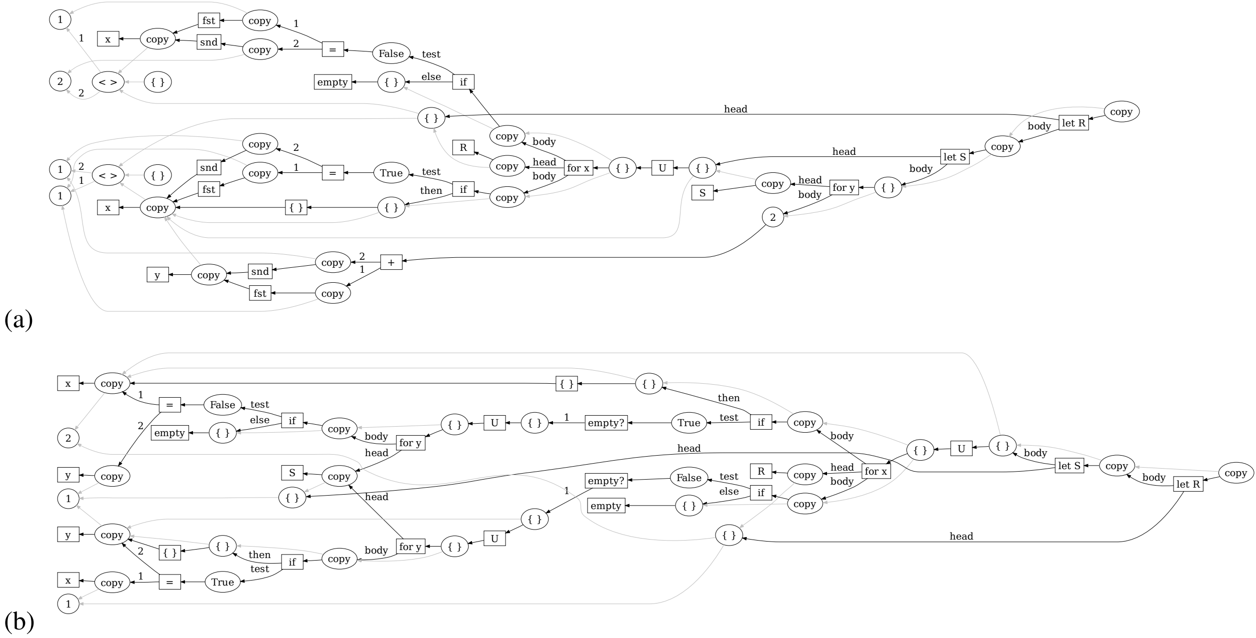
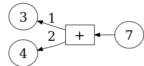


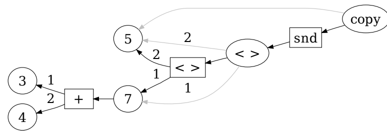
Figure 1: Examples (a) `SELECT a+b FROM R WHERE a=b`, where $R = \{(1,2),(1,1)\}$ and (b) `R MINUS S` where $\{R = \{1,2\}$ and $S = \{1\}$. Ovals are value nodes; boxes are evaluation nodes; gray edges are value-graph edges.

value or evaluation labels, such that: 1. $V = V_V \uplus V_E$, 2. prov is a partial injective function from V_V to V_E so that each evaluation node e has a unique value node $val(e)$, 3. G is a value graph if we disregard the evaluation graph structure, and 4. G is an evaluation graph if we merge each pair of nodes $(e, val(e))$ and disregard the value structure. In the following examples, by convention, we highlight parts of the input expression that are considered "inputs" using gray boxes.

Here is a simple example, showing the computation $3 + 4 = 7$:



Here is a more complicated example, showing the evaluation of expression $\pi_2(3+4, 5) = 5$ involving constructing a pair and then selecting the second argument:



Finally, here is a larger example demonstrating let-binding, showing the evaluation of an expression let $x = 3$ in let $y = 4$ in $x * x + y * y$.

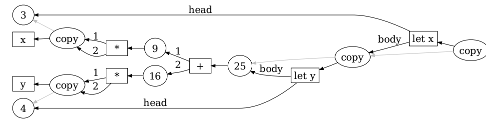
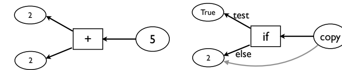


Figure 1(a) and (b) are two larger examples, corresponding to simple SQL queries.

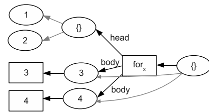
Given a provenance graph G and an evaluation node e, there is a natural notion of (G, e) being *locally consistent*, with the intuition that the computation depicted in the part of the evaluation graph reachable from e matches the assignment of value nodes to evaluation nodes by the val-edges. The graphs used as examples so far are all consistent in this sense; however, the following graphs are inconsistent:



The left example is obviously silly: the claimed result of a function should be consistent with the function's meaning. The right example is more subtle: the labels of the branches in conditional nodes need to match the boolean value of the test.

Moreover, we expect the evaluation graph to be globally consistent, in the sense that the whole trace is an "unfolding" of the evaluation of a particular expression e. All of the graphs we have seen so far are globally con-

sistent with an expression. However, the following graph is globally inconsistent:



The inconsistency here is between the two evaluation bodies of the for-loop: the body of a comprehension cannot be both a constant 3 and a constant 4. In a globally consistent graph the control flow leading to different values for different iterations must be made explicit.

It is possible to enforce local consistency using first-order constraints on the provenance graph. Moreover, global consistency can also be defined using first-order constraints by induction on the structure of expressions e. We omit the actual constraints due to space limits.

4 The provenance graph semantics

In this section we will show how to construct a consistent provenance graph by evaluating DFL expressions to construct both a value and evaluation graph, with appropriate links. Figure 2 illustrates the semantics schematically using some graph rewriting rules, where the rounded boxes describe the expression structure. Each rule schematically shows how an expression locally evaluates to values and expression nodes. These rules can be applied to build a provenance graph "bottom-up".

4.1 An executable definition

Defining an algorithm to actually construct a provenance graph can be tricky due to the large number of details we need to manage. We present a definition using Haskell [18], a functional language that provides sophisticated facilities for defining side-effecting operations such as those involved in building a graph. Using these features we can define a function that traverses an expression, evaluates it and constructs the associated provenance graph in only a few hundred lines of code. Haskell programs are precise definition that is still relatively readable and clear. Moreover, it is an executable specification that can be used to generate small examples and experiment with alternative definitions.

In general while evaluating an expression with free variables, we need to keep track of the values associated with variables, so we will introduce a little more notation to help with this bookkeeping.

For a finite set X of variables, a *value assignment* for X is a mapping $\sigma : X \to \mathsf{Val}$ that assigns to each $x \in X$ a complex value. We can model value assignments in a provenance graph G as follows. Let $\gamma : X \to G_E$ be a function mapping variable names to variable nodes of

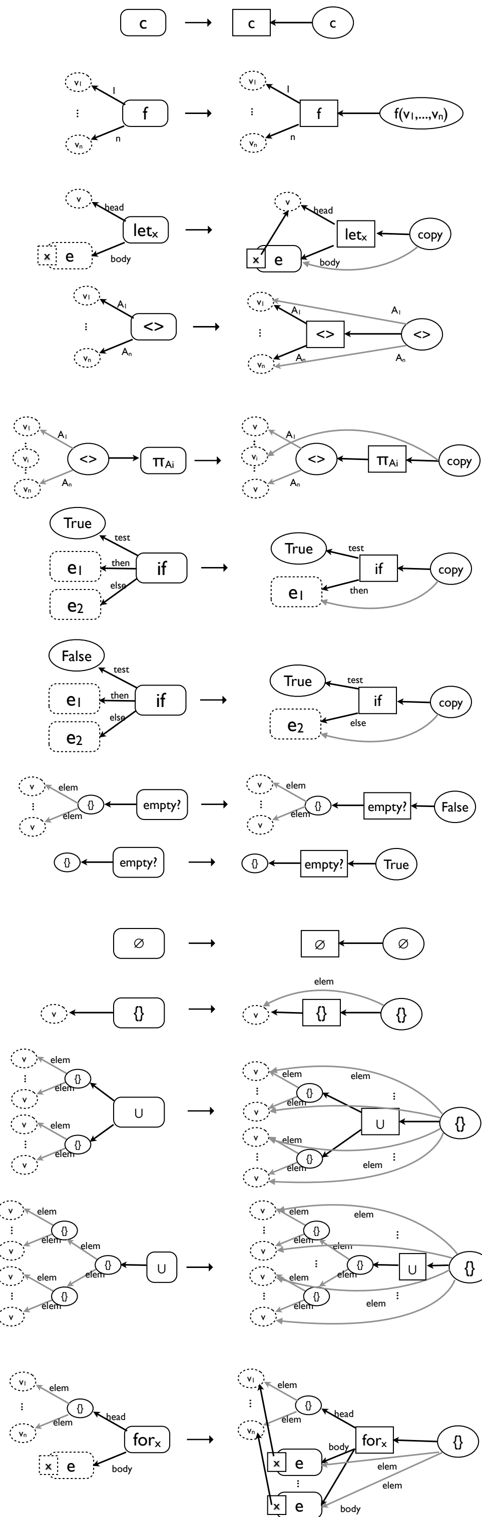


Figure 2: Graph rewriting rules for constructing provenance graphs

the evaluation graph of G, such that for each $x \in X$, we have $\mathsf{lab}_x(\gamma(x))$. Then for each x, the value node $\gamma(x)$ corresponds to a complex value. In particular, given an ordinary assignment $\sigma : X \to \mathsf{Val}$, we can always construct a graph G that defines value nodes for the values of $\sigma(x)$ and whose evaluation nodes correspond exactly to the variables in X.

In Haskell, it is more convenient to represent the graph as a collection of finite maps from nodes to datatypes that we shall call *constructors*. In our implementation, a single node will correspond to a pair (m, n) of an expression and value node in the previous development. We map each node to a value constructor and optionally an evaluation constructor. Constructors encode both the node and edge labels. For example, we use a constructor `EIf True n1 n2` to represent an if node with test-edge to n_1, a test value of True, and then-edge to n_2. This approach builds many of the basic validity properties of provenance graphs into the Haskell type system, making it easier to avoid trivial bugs. Of course, the constructor-based graph can be translated to the explicitly edge-labeled graphs used earlier. Figure 8 shows the basic datatypes for nodes, variables, contexts, and evaluation and value constructors in Haskell. Note that for simplicity we use Haskell's built-in list type for collections; we also restrict attention to ordered pairs rather than general records. These differences are inessential.

We also employ a feature of Haskell called *monads* [21] to structure the computation of the provenance graph for a given expression. Basically, a monad is a generic type $M\ a$. A value of type $M\ a$ is a computation that produces a value of type a and may have some side-effects. Because Haskell is a pure functional language, all side-effects need to be encapsulated within a monad. Monads can also handle contextual information such as tracking the current values of variables.

Our monad will employ the type:

```
type M a = Ctx -> Int -> Graph ->
                (Int,Graph,a)
```

Here, Ctx represents the variable context, the Int parameter/result is a counter used for generating fresh node ids, and the Graph parameter/result is the graph being built. The definition of the monad type and its operations in Haskell is slightly different for technical reasons. For presentation reasons we suppress these differences.

A monad is always equipped with two operations, here called "$\gg=$" (or "bind") and "return". The "return" operation simply takes a value of type a and produces a monad returning that value:

$$\begin{aligned} \mathsf{return} &:: a \to M\ a \\ \mathsf{return}\ a &= \lambda\gamma.\lambda i.\lambda G.(i, G, a) \end{aligned}$$

Furthermore, the "bind" operation takes an $M\ a$ (i.e., a computation producing values of type a) and a function from a to $M\ b$ and produces a computation $M\ b$ returning a value of type b. Its definition is as follows:

$$\begin{aligned} \gg= &:: M\ a \to (a \to M\ b) \to M\ b \\ f \gg= g &= \lambda\gamma.\lambda i.\lambda G. \\ &\quad\ \mathsf{let}\ (i', G', a) = f\ \gamma\ i\ G\ \mathsf{in}\ (g\ a)\ \gamma\ i'\ G' \end{aligned}$$

Finally, we define a number of operations that allow us to read the current state of the computation or perform a side-effecting operation. We give some examples in detail and then just describe the remaining operations.

To read the current value of a variable in the context, we define the lookup operation:

$$\begin{aligned} \mathsf{lookup} &:: \mathsf{Var} \to M\ \mathsf{Node} \\ \mathsf{lookup}\ x &= \lambda\gamma.\lambda i.\lambda G.(i, G, \gamma(x)) \end{aligned}$$

To create a fresh node, we define the following monadic operation that creates a new node using the current index and increments the index:

$$\begin{aligned} \mathsf{fresh} &:: M\ \mathsf{Node} \\ \mathsf{fresh} &= \lambda\gamma.\lambda i.\lambda G.(i+1, G, \mathsf{Node}(i)) \end{aligned}$$

Similar operations can be defined to access the current context, add a variable binding to the context, and so on. These operations are shown in Figure 3. We also include monadic versions of primitive functions (interp) and complex operations such as flattening ($\bigcup$). The operation link $econ\ vcon$ extends the graph with a new node n bound to the constructors $econ$ and $vcon$, returning n.

Figures 4, 6 and 7 show how to evaluate an expression in a provenance graph, producing a value node in an augmented provenance graph. The helper functions $[\![-]\!]^\dagger$ and $[\![-]\!]^*$ shown in Figure 5 help simplify the definition. First, $[\![e]\!]^\dagger$ simply evaluates an expression e to its value node and also returns the node's constructor. Second, $[\![e]\!]^*_{x \in vs}$ evaluates e repeatedly, with x bound in turn to each element of vs. The examples in Figure 1 were generated using the Haskell implementation and the `graphviz` Unix tool[1].

5 Querying the provenance graph

The provenance graph is a relational structure, and as such there are a wide variety of languages available for querying the graph, ranging from simple path or reachability queries, to SQL-like relational queries, to more expressive languages supporting recursive queries, such as Datalog. Thus, in a sense the problem of querying

[1] http://www.graphviz.org

```
data Node  = Node Int deriving (Eq,Ord,Show)
type Var   = String
type Ctx   = Var -> Node
data VCon = VInt Int | VBool Bool | VPair Node Node | VSet [Node] | VCopy Node
data ECon = EInt Int | EFun String [Node] | ELet Var Node Node | EVar Var
          | EPair Node Node | EProj Int Node | EBool Bool | EIf Bool Node Node
          | EEmpty | ESng Node | EUnion Node Node
          | EFor Node Var [Node] | EFlatten Node
data Graph = Graph {emap :: Map Node (Maybe ECon), vmap :: Map Node VCon}
```

Figure 8: Haskell code defining provenance graphs. The type `Map a b` consists of finite maps from type `a` to `b`, from the Haskell standard library.

$$
\begin{array}{rcl}
\mathsf{bindVar} & :: & \mathsf{Var} \to \mathsf{Node} \to M\ a \to M\ a \\
\mathsf{getVCon} & :: & \mathsf{Node} \to M\ \mathsf{VCon} \\
\mathsf{link} & :: & \mathsf{ECon} \to \mathsf{VCon} \to M\ \mathsf{Node} \\
\mathsf{interp} & :: & \mathsf{String} \to \mathsf{VCon} \to M\ \mathsf{VCon} \\
\mathsf{flatten} & :: & [\mathsf{Node}] \to M\ [\mathsf{Node}]
\end{array}
$$

Figure 3: Graph monad operations

the provenance graph is already solved by known techniques for querying arbitrary graphs that happen to be provenance.

However, it still seems to be a challenge to define known forms of provenance in databases in terms of provenance graphs. In this section, we sketch preliminary ideas towards this goal, using Datalog over the raw provenance graphs.

In order to define forms of where-provenance and why-provenance, we need a partial orders on evaluation nodes. We start by ordering the evaluation nodes in G based on the following two orders:

1. $Child(n_1, n_2)$ if $n_2 \xrightarrow{l} n_1$ (i.e., there is an edge from n_2 to n_1)
2. $Left(n_1, n_2)$ if
   - n_1 is the test-node of a conditional and n_2 is one of the branches
   - n_1 is the head-node of a let or for-node n and n_2 is a body-node of n

The partial order *Before* is defined as follows:

$$
\begin{array}{rcl}
Below(e,e) & \leftarrow & Eval(e) \\
Below(e,f) & \leftarrow & Below(e,g), Child(g,f) \\
Before(e,f) & \leftarrow & Below(e,f) \\
Before(e,f) & \leftarrow & Before(e,e'), Below(e',g), \\
 & & Left(g,h), Before(f,h)
\end{array}
$$

where in the first rule we ensure safety by constraining e

$$
\begin{array}{rcl}
[\![-]\!] & :: & \mathsf{Expr} \to M\ \mathsf{Node} \\
[\![x]\!] & = & \mathsf{lookup}\ x \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] & = & [\![e_1]\!] \gg\!= \lambda n_1. \\
 & & \mathsf{bindVar}\ x\ n_1\ [\![e_2]\!] \gg\!= \lambda n_2. \\
 & & \mathsf{link}(\mathsf{ELet}\ x\ n_1\ n_2)\ (\mathsf{VCopy}\ n_2) \\
[\![i]\!] & = & \mathsf{link}(\mathsf{EInt}\ i)\ (\mathsf{VInt}\ i) \\
[\![f(e)]\!] & = & [\![e]\!]^\dagger \gg\!= \lambda(n,v). \\
 & & \mathsf{interp}_f(v) \gg\!= \lambda v'. \\
 & & \mathsf{link}(\mathsf{EFun}\ f\ (n))\ (v')
\end{array}
$$

Figure 4: Monadic semantics for building provenance graphs, part 1 (variables, let, primitive functions)

$$
\begin{array}{rcl}
[\![-]\!]^\dagger & :: & \mathsf{Expr} \to M\ (\mathsf{Node}, \mathsf{VCon}) \\
[\![e]\!]^\dagger & = & [\![e]\!] \gg\!= \lambda n. \\
 & & \mathsf{getVCon}(n) \gg\!= \lambda v. \\
 & & \mathsf{return}\ (n, v) \\
[\![-]\!]^*_{x \in []} & = & \mathsf{return}\ [] \\
[\![e]\!]^*_{x \in n_0 : ns} & = & \mathsf{bindVar}\ x\ n_0\ [\![e]\!] \gg\!= \lambda n'_0. \\
 & & [\![e]\!]^*_{x \in ns} \gg\!= \lambda ns'. \\
 & & \mathsf{return}\ (n'_0 : ns')
\end{array}
$$

Figure 5: Helper functions $[\![e]\!]^\dagger$ and $[\![e]\!]^*_{x \in vs}$.

to be an evaluation node using a predicate $Eval(e)$ that lists all evaluation nodes.

We also define the *Copy* relation on value nodes as the reflexive, transitive closure of the copy edge relation. We use the predicate $Value(n)$ in the first rule to constrain n to be a value node, ensuring safety:

$$
\begin{array}{rcl}
Copy(n,n) & \leftarrow & Value(n) \\
Copy(n,n') & \leftarrow & Copy(n,n''), n'' \xrightarrow{\mathsf{copy}} n'
\end{array}
$$

We will define where-provenance and why-provenance queries on pairs (e, v) such that v is reachable by a directed path (possibly including copy-links) from e. We will call such pairs *instances*.

$$
\begin{array}{rcl}
[\![(e_1, e_2)]\!] & = & [\![e_1]\!] \ggg\!= \lambda n_1. \\
 & & [\![e_2]\!] \ggg\!= \lambda n_2. \\
 & & \mathsf{link}(\mathsf{EPair}(n_1, n_2))\ (\mathsf{VPair}(n_1, n_2)) \\
[\![\pi_i(e)]\!] & = & [\![e]\!]^\dagger \ggg\!= \lambda(n, \mathsf{VPair}(n_1, n_2)) \\
 & & \mathsf{link}(\mathsf{EProj}\ i\ n)\ (\mathsf{VCopy}\ n_i) \\
[\![b]\!] & = & \mathsf{link}(\mathsf{EBool}\ b)\ (\mathsf{VBool}\ b) \\
[\![\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!] & = & [\![e]\!]^\dagger \ggg\!= \lambda(n, \mathsf{VBool}\ b). \\
 & & \mathsf{if}\ b \\
 & & \ \mathsf{then}\ [\![e_1]\!] \ggg\!= \lambda n'. \\
 & & \quad \mathsf{link}(\mathsf{EIf}\ \mathsf{True}\ n\ n')\ (\mathsf{VCopy}\ n') \\
 & & \ \mathsf{else}\ [\![e_2]\!] \ggg\!= \lambda n'. \\
 & & \quad \mathsf{link}(\mathsf{EIf}\ \mathsf{False}\ n\ n')\ (\mathsf{VCopy}\ n')
\end{array}
$$

Figure 6: Monadic semantics for building provenance graphs, part 2 (pairs, conditionals)

$$
\begin{array}{rcl}
[\![\emptyset]\!] & = & \mathsf{link}(\mathsf{EEmpty})\ (\mathsf{VSet}\ []) \\
[\![\{e\}]\!] & = & [\![e]\!] \ggg\!= \lambda n. \\
 & & \mathsf{link}(\mathsf{ESng}\ n)\ (\mathsf{VSet}\ [n]) \\
[\![e_1 \cup e_2]\!] & = & [\![e_1]\!]^\dagger \ggg\!= \lambda(n_1, \mathsf{VSet}(vs_1)). \\
 & & [\![e_2]\!]^\dagger \ggg\!= \lambda(n_2, \mathsf{VSet}(vs_2)). \\
 & & \mathsf{link}(\mathsf{EUnion}\ n_1\ n_2)\ (\mathsf{VSet}(vs_1 +\!\!+ vs_2)) \\
[\![\{e \mid x \in e_0\}]\!] & = & [\![e_0]\!] \ggg\!= \lambda(n_0, \mathsf{VSet}(ns)). \\
 & & [\![e]\!]^*_{x \in vs} \ggg\!= \lambda ns'. \\
 & & \mathsf{link}(\mathsf{EFor}\ n_0\ x\ ns')\ (\mathsf{VSet}\ ns') \\
[\![\bigcup e]\!] & = & [\![e]\!] \ggg\!= \lambda(n, \mathsf{VSet}(ns)). \\
 & & \mathsf{flatten}(ns) \ggg\!= \lambda ns'. \\
 & & \mathsf{link}\ (\mathsf{EFlatten}\ n)\ (\mathsf{VSet}\ ns')
\end{array}
$$

Figure 7: Monadic semantics for building provenance graphs, part 3 (collections)

5.1 Where-provenance

We now can define a form of where-provenance on instances (e, v) as follows:

$$
\begin{array}{rcl}
\mathit{Where}((e_1, v_1), (e_2, v_2)) & \leftarrow & \mathit{Before}(e_1, e_2), \\
 & & \mathit{Copy}(v_2, v_1)
\end{array}
$$

Intuitively, this says that the where-provenance of the value v_2 returned by the evaluation ending at e_2 is the same as that of the value v_1 returned by e_1. Clearly, there should be a unique least (e_1, v_1) with respect to Before so we can define the where-provenance as that instance (e_1, v_1). This definition relies on the fact that our provenance graph already corresponds closely to the high-level view of where-provenance defined via annotation-propagation in previous work [3, 2]. One important difference is that here, we can refer to intermediate steps in the provenance graph.

5.2 Why-provenance

Why-provenance was defined in [3] using witnesses. There, a witness to the existence of a part p of the output of a query Q on input data d was defined as a subtree of the input d such that rerunning Q on the subtree still produces output part p. We generalize this idea as follows. Let G be a provenance graph with a distinguished evaluation node r whose value is v. Let U be a connected subset of the result value nodes that contains v. Then a *witness* to U in G is a consistent subgraph of G that contains r and U. The *why-provenance* of V is then the set of all minimal witnesses to U in G.

Alternatively, we could give a low-level definition that traverses the graph to construct a witness starting from a set of output value nodes, following similar lines to the low-level definition of Where above. We omit the details; the main differences are in rules for conditionals and primitive functions where we continue tracking the dependencies of the test or input values respectively.

5.3 Discussion

This discussion suggests a number of interesting observations and questions which we will not attempt to resolve here, including:

- The first definition of Where appears equivalent to the where-provenance model of [2]. Can we make this connection precise? Likewise, can we formally relate the why-provenance subgraphs to definitions of why-provenance or lineage for NRC (e.g. [11])?
- The above definitions characterize where-provenance structurally by following a chain of copies from the input to the output (or vice versa). This exhibits a symmetry between querying the provenance graph "forward" vs. "backward". Is this a unique feature of where-provenance or are there other provenance queries that have this property?
- In some prior work (e.g. [2]), forms of provenance have been defined by translating NRC queries to queries that explicitly manage their own provenance information. Is this possible for provenance graphs?
- We explored a number of design choices that could be revisited. For example, we considered treating let transparently and avoiding using explicitly labeled variable nodes. Another controversial choice was the use of copy-links rather than directly sharing value nodes. Are these differences important, and how can we determine this?
- How should updates be modeled?

We conclude this section with some sheer speculation.

Sensible provenance queries Consider all queries on provenance graphs expressible in, say, relational calculus or Datalog. Clearly, this includes many strange queries

that test properties of the graph that seem irrelevant to provenance. For example, a query might select all value nodes that appear exactly three times in the graph, or all graphs that contain fewer than seventeen nodes, or all graphs such that function f is called an even number of times. Is there a natural characterization of "sensible" provenance queries?

Provenance query answerability Consider the following problem: given two provenance graph queries q_1, q_2, can we answer q_2 using the results of q_1 (without access to the original graph, expression or input data)? If so, it is reasonable to say that q_1 is more general than q_2. This problem is an instance of the problem of answering queries using views, which has been studied for relational and XML databases. Can these results be applied or adapted to provenance queries?

Query rewriting and provenance query equivalence Many equivalent queries become inequivalent in the provenance graph semantics — for example, union is no longer commutative. Does this matter, or is it acceptable to optimize queries using ordinary equivalence rules? Under what conditions is this reasonable?

Efficient provenance querying Provenance graphs may be too large to construct or retain in practice. Given a provenance graph query q, can we compute the answer to q more efficiently and without materializing the full provenance graph? Can we compute provenance graphs "lazily" or for just a part of the result, in response to a query, rather than "eagerly" computing the whole graph?

6 Related Work

We have attempted to remain close to de facto standards for visualizing provenance as graphs, particularly the Open Provenance Model [22], which distinguishes between "process" (evaluation) and "artifact" (value) nodes. However, we have not included other aspects of OPM and have not tried to make our graphs fit OPM exactly. Collections are not modeled directly in OPM 1.0, although a proposal for describing the provenance of collections in OPM is under development [15]. OPM is an exchange and representation format for provenance information and so some of the the semantic issues we investigate are outside its scope.

Semantics and models of provenance have been studied in formal detail for some systems. Sroka et al. [25] develop a semantics for Taverna workflows based on a core language similar to NRC over lists, but including implicit coercions from elements to collections and also incorporating operations such as zip that are not expressible in plain NRC. Missier et al. [20] discuss lightweight lineage annotations for Taverna workflows but does not fully detail how to the full language presented in [25].

Graphical notations for provenance have been used extensively in many systems. Recent work of interest in some of these systems includes work on consistency and optimization for user views (abstractions) of provenance graphs [26] and efficiently representing and storing provenance graphs over nested collections [1]. Again, however, these papers focus on structural aspects of provenance graphs as produced by some workflow system, whereas we are studying the relationship between a provenance graph and the computation performed by the system.

Our work is inspired partly by "provenance traces" [6], an approach to provenance for NRC in which evaluation of expressions yields both a value and a "trace", or detailed record of evaluation. Traces in that work are complete in the sense that they can be used to replay the computation under arbitrary changes to the input; our provenance graphs do not try to support replay under arbitrary changes, but may therefore be more compact. Moreover, our provenance graphs explicate the relationship between workflow and database provenance models, a question not addressed in [6].

Another closely related line of work is on the NRC-based dataflow language DFL, starting with Hidders et al. [17]. Hidders et al. [16] developed a *run semantics* for dataflow calculus programs along with a sketch of how to extract provenance from runs. Subsequently Kwasnikowska and van den Bussche [19] showed how to represent the run semantics using OPM. This work also modeled nondeterministic service (external function) calls and modeled procedures using OPM accounts. These refinements have been left out in our presentation but can easily be handled. Our work improves on this approach by directly defining a provenance graph semantics that seems closer to typical workflow provenance, and avoiding some technical complications involved in the previous approach, particularly the use of paths to refer to parts of values and expressions indirectly. On the other hand, our pervasive use of freshly generated graph nodes introduces complications of its own, which we have addressed using the Haskell monadic programming style. We also discuss specific provenance queries defined using Datalog queries on the provenance graph, although we only have preliminary results and many open questions are evident.

Although the semantics we propose here may not be universally acceptable or usable off-the-shelf in these other settings, we believe the methods we outline are re-usable. In particular, the idea of using Haskell-style monads to define a precise semantics is very flexible, since monads can incorporate many other kinds of side-effects besides fresh node identifier generation. It may be worthwhile to model the provenance semantics of more realistic workflow languages in Haskell, facilitating di-

rect comparisons of the behavior and provenance of different workflow languages.

7 Conclusions

Although provenance has been studied in both database and workflow settings for more than a decade, little has been done to relate the approaches. In this paper we make two contributions in this direction, in the context of the previously-introduced dataflow calculus DFL. First, we detail a semantics that evaluates dataflow calculus expressions to provenance graphs containing values, evaluation nodes, and links showing how the expression evaluated, and we discuss interesting kinds of queries on the resulting structure, related to where-provenance and why-provenance in databases. Second, we present a concise and precise formal version of this model implemented using Haskell, a high-level functional language.

We believe our work helps bridge a gap between the theoretical approaches that so far have largely been employed for database provenance and the practical, but sometimes loosely-specified techniques developed in workflow systems. We also identified a number of interesting research questions concerning where- and why-provenance queries over the provenance graph and their relationships to these forms of provenance in databases. We are investigating these in ongoing work. Although there is still room for debate about the particular design choices made in our approach, our formal model at least makes it easier to hold such a debate, and to experiment with alternatives.

Acknowledgments We have discussed ideas related to this work with Amal Ahmed. This work has been supported by EPSRC grant EP/F028288/1.

References

[1] ANAND, M. K., BOWERS, S., MCPHILLIPS, T., AND LUDÄSCHER, B. Efficient provenance storage over nested data collections. In *EDBT* (New York, NY, USA, 2009), ACM, pp. 958–969.

[2] BUNEMAN, P., CHENEY, J., AND VANSUMMEREN, S. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems 33*, 4 (November 2008), 28.

[3] BUNEMAN, P., KHANNA, S., AND TAN, W. Why and where: A characterization of data provenance. In *ICDT* (2001), no. 1973 in LNCS, Springer, pp. 316–330.

[4] BUNEMAN, P., KHANNA, S., AND TAN, W. On propagation of deletions and annotations through views. In *PODS* (2002), pp. 150–158.

[5] BUNEMAN, P., NAQVI, S. A., TANNEN, V., AND WONG, L. Principles of programming with complex objects and collection types. *Theor. Comp. Sci. 149*, 1 (1995), 3–48.

[6] CHENEY, J., ACAR, U. A., AND AHMED, A. Provenance traces. *CoRR abs/0812.0564* (2008).

[7] CHENEY, J., AHMED, A., AND ACAR, U. A. Provenance as dependency analysis. In *DBPL 2007* (Vienna, Austria, September 2007), M. Arenas and M. I. Schwartzbach, Eds., no. 4797 in LNCS, Springer-Verlag, pp. 139–153.

[8] CHENEY, J., CHITICARIU, L., AND TAN, W. C. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases 1*, 4 (2009), 379–474.

[9] CUI, Y., WIDOM, J., AND WIENER, J. L. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst. 25*, 2 (2000), 179–227.

[10] FOSTER, I., VOCKLER, J., WILDE, M., AND ZHAO, Y. Chimera: A virtual data system for representing, querying, and automating data derivation. In *SSDBM* (July 2002), pp. 1–10.

[11] FOSTER, J. N., GREEN, T. J., AND TANNEN, V. Annotated XML: queries and provenance. In *PODS* (2008), pp. 271–280.

[12] FREIRE, J., KOOP, D., AND MOREAU, L., Eds. *IPAW* (2008), vol. 5272 of *Lecture Notes in Computer Science*, Springer.

[13] GREEN, T. J., KARVOUNARAKIS, G., AND TANNEN, V. Provenance semirings. In *PODS* (2007), ACM, pp. 31–40.

[14] GROTH, P., LUCK, M., AND MOREAU, L. A protocol for recording provenance in service-oriented grids. In *OPODIS '04* (2004).

[15] GROTH, P., MILES, S., MISSIER, P., AND MOREAU, L. A proposal for handling collections in the open provenance model, 2009.

[16] HIDDERS, J., KWASNIKOWSKA, N., SROKA, J., TYSZKIEWICZ, J., AND VAN DEN BUSSCHE, J. A formal model of dataflow repositories. In *DILS* (2007), vol. 4544 of *LNCS*, Springer, pp. 105–121.

[17] HIDDERS, J., KWASNIKOWSKA, N., SROKA, J., TYSZKIEWICZ, J., AND VAN DEN BUSSCHE, J. DFL: A dataflow language based on petri nets and nested relational calculus. *Inf. Syst. 33*, 3 (2008), 261–284.

[18] HUTTON, G. *Programming in Haskell*. Cambridge University Press, 2007.

[19] KWASNIKOWSKA, N., AND VAN DEN BUSSCHE, J. Mapping the NRC dataflow model to the open provenance model. In Freire et al. [12], pp. 3–16.

[20] MISSIER, P., BELHAJJAME, K., ZHAO, J., ROOS, M., AND GOBLE, C. A. Data lineage model for taverna workflows with lightweight annotation requirements. In Freire et al. [12], pp. 17–30.

[21] MOGGI, E. Notions of computation and monads. *Inf. Comput. 93*, 1 (1991), 55–92.

[22] MOREAU, L., FREIRE, J., FUTRELLE, J., MCGRATH, R. E., MYERS, J., AND PAULSON, P. The open provenance model: An overview. In Freire et al. [12], pp. 323–326.

[23] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *USENIX ATC* (June 2006), USENIX, pp. 43–56.

[24] SIMMHAN, Y. L., PLALE, B., AND GANNON, D. Karma2: Provenance management for data-driven workflows. *Int. J. Web Service Res. 5*, 2 (2008), 1–22.

[25] SROKA, J., HIDDERS, J., MISSIER, P., AND GOBLE, C. A formal semantics for the taverna 2 workflow model. *Journal of Computer and System Sciences In Press, Corrected Proof* (2009).

[26] SUN, P., LIU, Z., DAVIDSON, S. B., AND CHEN, Y. Detecting and resolving unsound workflow views for correct provenance analysis. In *SIGMOD* (New York, NY, USA, 2009), ACM, pp. 549–562.