Under consideration for publication in Math. Struct. in Comp. Science

Provenance as Dependency Analysis[†]

James Cheney[1] [‡] and Amal Ahmed[2] and Umut A. Acar[3] [§]

[1] *University of Edinburgh*
[2] *Indiana University*
[3] *Max-Planck Institute for Software Systems*

Received 5 November 2009

Provenance is information recording the source, derivation, or history of some information. Provenance tracking has been studied in a variety of settings, particularly database management systems; however, although many candidate definitions of provenance have been proposed, the mathematical or semantic foundations of data provenance have received comparatively little attention. In this article, we argue that dependency analysis techniques familiar from program analysis and program slicing provide a formal foundation for forms of provenance that are intended to show how (part of) the output of a query *depends* on (parts of) its input. We introduce a semantic characterization of such *dependency provenance* for a core database query language, show that minimal dependency provenance is not computable, and provide dynamic and static approximation techniques. We also discuss preliminary implementation experience with using dependency provenance to compute data slices, or summaries of the parts of the input relevant to a given part of the output.

1. Introduction

Provenance is information about the origin, ownership, influences upon, or other historical or contextual information about an object. Such information has many applications, including evaluating integrity or authenticity claims, detecting and repairing errors, and memoizing and caching the results of computations such as scientific workflows [Lynch, 2000, Bose and Frew, 2005, Simmhan et al., 2005]. Provenance is particularly important in scientific computation and record-keeping, since it is considered essential for ensuring the repeatability of experiments and judging the scientific value of their results [Buneman et al., 2008a].

Most computer systems provide simple forms of provenance, such as the timestamp and ownership metadata in file systems, system logs, and version control systems. Richer provenance tracking techniques have been studied in a variety of settings, including databases [Cui et al., 2000, Buneman et al., 2001, 2008b, Foster et al., 2008, Green et al., 2007], file systems [Muniswamy-Reddy et al., 2006], and scientific workflows [Bose and Frew, 2005, Simmhan et al., 2005]. Although a wide variety of design points have been explored, there is relatively little

[†] This paper is a revised and extended version of [Cheney et al., 2007].

[‡] Cheney is supported by EPSRC grant GR/S63205/01 and a Royal Society University Research Fellowship.

[§] Acar is supported by gifts from Intel and Microsoft Research.

understanding of the relationships among techniques or of the design considerations that should be taken into account when developing or evaluating an approach to provenance. The mathematical or semantic foundations of data provenance have received comparatively little attention. Most prior approaches have invoked intuitive concepts such as *contribution*, *influence*, and *relevance* as motivation for their definitions of provenance. These intuitions suggest definitions that appear adequate for simple (e.g. conjunctive) relational queries, but are difficult to extend to handle more complex queries involving subqueries, negation, grouping, or aggregation.

However, these intuitions have also motivated rigorous approaches to seemingly quite different problems, such as aiding debugging via *program slicing* [Biswas, 1997, Field and Tip, 1998, Weiser, 1981], supporting efficient memoization and caching [Abadi et al., 1996, Acar et al., 2003], and improving program security using information flow analysis [Sabelfeld and Myers, 2003]. As Abadi et al. [1999] have argued, slicing, information flow, and several other program analysis techniques can all be understood in terms of dependence. In this article, we argue that these dependency analysis and slicing techniques familiar from programming languages provide a suitable foundation for an interesting class of provenance techniques.

To illustrate our approach, consider the input data shown in Figure 1(a) and the SQL query in Figure 1(b) which calculates the average molecular weights of proteins involved in each reaction. The result of this query is shown in Figure 1(c). The intuitive meaning of the SQL query is to find all combinations of rows from the three tables Protein, EnzymaticReaction, and Reaction such that the conditions in the WHERE-clause hold, then group the results by the Name field, while averaging the MW (molecular weight) field values and returning them in the AvgMW field.

Since the MW field contains the molecular weight of a protein, it is clearly an error for the italicized value in the result to be negative. To track down the source of the error, it would be helpful to know which parts of the input contributed to, or were relevant to, the erroneous part of the output. We can formalize this intuition by saying that a part of the output depends on a part of the input if a change to the input part may result in a change to the output part. This is analogous to the notion of dependence underlying program slicing [Weiser, 1981], a debugging aid that identifies the parts of a program on which a program output may depend.

In this example, the input field values that the erroneous output AvgMW-value depends on are highlighted in bold. The dependencies include the two summed MW values and the ID fields which are compared by the selection and grouping query. These ID fields must be included because a change to any one of them could result in a change to the italicized output value—for example, changing the occurrence of p_4 in table EnzymaticReaction would change the average molecular weight in the second row. On the other hand, the names of the proteins and reactions are irrelevant to the output AvgMW—no changes to these parts can have any effect on the italicized value, and so we can safely ignore these parts when looking for the source of the error.

This example is simplistic, but the ability to concisely explain which parts of the input influence each part of the output becomes more important if we consider a large query to a realistic database with tens or hundreds of columns per table and thousands or millions of rows. Manually tracing the dependence information in such a setting would be prohibitively labor-intensive. Moreover, dependence information is useful for a variety of other applications, including estimating the *freshness* of data in a query result by aggregating timestamps on the relevant inputs, or transferring *quality* annotations provided by users from the outputs of a query back to the inputs.

For example, suppose that each part of the database is annotated with a timestamp. Given a

(a)

Protein			
ID	Name	**MW**	···
$\mathbf{p_1}$	thioredoxin	$\mathbf{11.8}$	···
$\mathbf{p_2}$	flavodoxin	19.7	···
$\mathbf{p_3}$	ferredoxin	12.3	···
$\mathbf{p_4}$	ArgR	$\mathbf{-700}$	···
$\mathbf{p_5}$	CheW	18.1	···
⋮	⋮	⋮	⋮

EnzymaticReaction	
PID	**RID**
$\mathbf{p_1}$	$\mathbf{r_1}$
$\mathbf{p_2}$	$\mathbf{r_1}$
$\mathbf{p_1}$	$\mathbf{r_2}$
$\mathbf{p_4}$	$\mathbf{r_2}$
$\mathbf{p_5}$	$\mathbf{r_3}$
⋮	⋮

Reaction		
ID	Name	···
$\mathbf{r_1}$	t-p + ATP = t d + ADP	···
$\mathbf{r_2}$	H_2O + an a p → p + a c	···
$\mathbf{r_3}$	D-r-5-p = D-r-5-p	···
$\mathbf{r_4}$	β-D-g-6-p = f-6-p	···
$\mathbf{r_5}$	p $4'$-p + ATP = d-CoA + d	···
⋮	⋮	⋮

(b)

```
SELECT   R.Name as Name, AVERAGE(P.MW) as AvgMW
FROM     Protein P, EnzymaticReaction ER, Reaction R
WHERE    P.ID = ER.ProteinID, ER.ReactionID = R.ID
GROUP BY R.Name
```

(c)

Name	*AvgMW*
t-p + ATP = t d + ADP	15.75
H_2O + an a p → p + a c	*-338.2*
D-r-5-p = D-r-5-p	18.1
⋮	⋮

Fig. 1. Example (a) input, (b) query, and (b) output data; input field names and values relevant to the *italicized* erroneous output field or value are highlighted in **bold**.

query, the dependence information shown in Figure 1 can be used to estimate the last modification time of the data relevant to each part of the output, by summarizing the set of timestamps of parts of the input contributing to an output part. Similarly, suppose that the system provides users with the ability to provide quality feedback in the form of star ratings. If a user flags the negative AvgMW value as being of low quality, this feedback can be propagated back to the underlying data according to the dependence information shown in Figure 1 and provided to the database maintainers who may find it useful in finding and correcting the error. In many cases, the user who finds the error may also be the database maintainer, but dependency information still seems useful as a debugging (or data cleaning) tool even if one has direct access to the data.

In this article, we argue that data dependence provides a solid semantic foundation for a provenance technique that highlights parts of the input on which each part of the output depend. We work in the setting of the *nested relational calculus* (NRC) [Buneman et al., 1994, 1995, Wong, 1996], a core language for database queries that is closely related to *monad algebra* [Wadler, 1992]. The NRC provides all of the expressiveness of popular query languages such as SQL, and includes *collection types* such as sets or multisets, equipped with union, comprehension, difference and equality operations. The NRC can also be extended to handle SQL's grouping and aggregation operations, and functions on basic types. We consider annotation-propagating semantics for such queries and define a property called *dependency-correctness*, which, intuitively, means that the provenance annotations produced by a query reflect how the output of the query may change if the input is changed.

There may be many possible dependency-correct annotation-propagating queries corresponding to an ordinary query. In general, it is preferable to minimize the annotations on the result, since this provides more precise dependency information. Unfortunately, as we shall show, minimal annotations are not computable. Instead, therefore, we develop dynamic and static techniques that produce dependency-correct annotations that are not necessarily minimal. We have implemented these techniques and found that they yield reasonable results on small-scale examples; the implementation was used to generate the results shown in Figure 1.

1.1. *Prior Work on Provenance*

We first review the relevant previous work on provenance and contrast it with our approach. We provide a detailed comparison with prior work on program slicing and information flow in Section 6.

1.1.1. *Provenance for database queries* Provenance for database queries has been studied by a number of researchers, beginning in the early 1990s [Buneman et al., 2001, 2002, 2008b, Cui et al., 2000, Green et al., 2007, Wang and Madnick, 1990, Woodruff and Stonebraker, 1997]. Recent research on annotations, uncertainty, and incomplete information [Benjelloun et al., 2006, Bhagwat et al., 2005, Geerts et al., 2006] has also drawn on these approaches to provenance; in particular, definitions of provenance have been used to justify annotation-propagation behaviors in these systems. We will focus on the differences between our work and the most recent work; please see Buneman et al. [2008a] and Cheney et al. [2009] for more complete discussion of research on provenance in databases.

Most prior work on provenance has focused on identifying information that explains *why* some data is present in the output of a query (or view) or *where* some data in the output was copied from in the input. However, satisfying semantic characterizations of these intuitions have proven elusive; indeed, many of the proposed definitions themselves have been unclear or ambiguous. Many proposed forms of provenance are sensitive to query rewriting, in that equivalent database queries may have different provenance behavior. This raises a number of troubling issues, since database systems typically rewrite queries modulo equivalence, so the provenance of a query may change as a result of query optimization. Also, in part because of the absence of clear formal definitions and foundations, these approaches have been difficult to generalize beyond monotone relational queries in a principled way.

In *why- and where-provenance*, introduced by Buneman et al. [2001], provenance is studied in a deterministic tree data model, in which each part of the database can be addressed by a unique path. Buneman et al. [2001] considered two forms of provenance: *why-provenance*, which consists of a set of witnesses, or subtrees of the input that suffice to explain a part of the output, and *where-provenance*, which consists of a *single* part of the input from which a given part of the output was copied. Both forms of provenance are sensitive to query rewriting in general, but Buneman et al. [2001] discussed normal forms for queries that avoid this problem.

Green et al. [2007] showed that relations with semiring-valued annotations on rows generalize several variations of the relational model, including set, bag, probabilistic, and incomplete information models, and identified a relationship between free semiring-valued relations and why-provenance. Foster et al. [2008] extended this approach to handle NRC queries and an unordered variant of XML. These approaches also appear orthogonal to our approach, and in additional only consider annotations at the level of elements of collections, not individual fields or collections, and they do not handle negation or aggregation.

Buneman et al. [2008b] introduced a model of where-provenance for the nested relational calculus. In their approach each part of the database is tagged with an optional annotation, or *color*; colors are propagated to the output so as to indicate where parts of the output have been copied from in the input. They studied the expressiveness of this model compared to queries that explicitly manipulate annotations. They also investigated where-provenance for updates, which we discuss in Section 1.1.2.

Our work is closest in spirit to the why-provenance and lineage techniques; however, in contrast to these techniques our approach annotates every part of the database and provides clear semantic guarantees and qualitatively useful provenance information in the presence of negation, grouping and aggregation.

1.1.2. *Provenance for database updates* Some recent work has generalized where-provenance to database updates [Buneman et al., 2006, 2008b], motivated by *curated* scientific databases that are updated frequently, often by (manual) copying from other sources. This work has focused on recording the external sources of the data in a database and tracking how the data has been rearranged within a database across successive versions. Accordingly, the provenance information provided by these approaches only connects data to exact copies in other locations, and does not track provenance through other operations. In this sense, it is similar to the where-provenance approach considered by Buneman et al. [2001] for database queries.

Our approach addresses an orthogonal issue, that of understanding how data in the result of a query depends on parts of the input; we therefore track provenance through copies as well as other forms of computation. Although our definition of dependency correctness could also be used for updates, it is not clear whether this yields a useful form of provenance, and we plan to investigate alternative dependency conditions that are more suitable for updates, using the update language employed in [Buneman et al., 2008b].

1.1.3. *Workflow provenance* Provenance has also been studied in geospatial and scientific computation [Bose and Frew, 2005, Foster and Moreau, 2006, Simmhan et al., 2005], particularly for *workflows* (visual programs written by scientists). In their simplest form (see e.g. the Provenance Challenge [Moreau et al., 2007]), workflows are essentially directed acyclic graphs (DAGs) representing a computation. For such DAG workflows, the provenance information that is typically stored is simply the workflow DAG, annotated with additional information, such as filenames and timestamps, describing the arguments that were used to compute the result of interest. This corresponds to a simple form of dependency tracking, although as far as we know no research on workflow provenance has explicitly drawn this connection.

However, many more sophisticated workflow programming models have been developed, involving concurrency and distributed computation. For these models, the appropriate correctness criteria for provenance tracking are much less clear. In fact, the ordinary semantics of these systems is not always clearly specified. One principled approach recently introduced by Hidders et al. [2007] defines provenance for the nested relational calculus augmented with additional function symbols that represent calls to scientific workflow components. Their approach has so far focused on defining provenance and not formulating or proving desirable correctness properties. We believe dependence analysis may provide an appropriate foundation for provenance in this and other workflow programming models.

1.2. *Contributions*

The main contribution of this article is the development of a semantic criterion called *dependency-correctness* that characterizes a form of provenance information we call *dependency provenance*. Dependency-correctness captures an intuition that provenance should link a part of the output to all parts of the input on which the output part depends, enabling us to make some predictions about the effects of changes to the input and to quickly identify source data that contributed to an error in the output.

Building on this framework, we show that (unsurprisingly) it is undecidable whether some dependency-correct provenance information is minimal, and proceed to develop computable dynamic and static techniques for conservatively approximating correct dependency provenance. These techniques, and their correctness proofs, are largely standard but the presence of database query language features and collection types introduces complications that have not been addressed before in work on information flow or program slicing.

1.3. *Organization*

The structure of the rest of this article is as follows. We review the syntax, type system and semantics of the nested relational calculus in Section 2. We then introduce (in Section 3) the annotation-propagation model, motivate and define dependency-correctness, and show that it is impossible to compute minimal dependency-correct annotations. In Section 4 we describe a dynamic *provenance-tracking* semantics that is dependency-correct. We also (Section 5) introduce a static, type-based *provenance analysis* which is less accurate than provenance tracking, but can be performed statically; we also prove its correctness relative to dynamic provenance tracking. We discuss experience with a prototype implementation in Section 6 and discuss future work and conclude in Section 7.

2. Background

We will provide a brief review of the *nested relational calculus* (NRC) [Buneman et al., 1995], a core database query language which is closely related to *monad algebra* [Wadler, 1992]. The nested relational calculus is a typed functional language with types τ of the form:

$$\tau \quad ::= \quad \mathsf{bool} \mid \mathsf{int} \mid \tau_1 \times \tau_2 \mid \{\tau\}$$

We consider base types bool and int, along with product types $\tau_1 \times \tau_2$ and *collection types* $\{\tau\}$. Collection types typically are taken to be monads equipped with an addition operator (sometimes called *ringads*); typical examples used in databases include lists, sets, or multisets, and in this article we consider finite multisets (also known as bags).

The expressions of our variant of NRC are as follows:

$$\begin{array}{rcl} e & ::= & x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid (e_1, e_2) \mid \pi_1(e) \mid \pi_2(e) \\ & \mid & b \mid \neg e \mid e_1 \wedge e_2 \mid e_1 \approx e_2 \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\ & \mid & i \mid e_1 + e_2 \mid \mathsf{sum}(e) \\ & \mid & \emptyset \mid \{e\} \mid e_1 \cup e_2 \mid e_1 - e_2 \mid \{e_2 \mid x \in e_1\} \mid \bigcup e \end{array}$$

Here, $i \in \mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$ denotes integer constants and $b \in \mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ denotes Boolean constants. The bag operations include $\emptyset$, the constant empty multiset; singletons $\{e\}$; multiset union $\cup$, difference $-$, and comprehension $\{e_2 \mid x \in e_1\}$; and flattening $\bigcup e$. By convention, we write $\{e_1, \ldots, e_n\}$ as syntactic sugar for $\{e_1\} \cup \cdots \cup \{e_n\}$. Finally, we include sum, a typical *aggregation* operation, which adds together all of the elements of a multiset and produces a value; e.g. $\mathsf{sum}\{1, 2, 3\} = 6$. By convention, we take $\mathsf{sum}(\emptyset) = 0$. We syntactically distinguish between NRC's equality operation $\approx$ and mathematical equality $=$.

NRC expressions can be typechecked using standard techniques. Contexts Γ are lists of pairs of variables and types $x_1 : \tau_1, \ldots, x_n : \tau_n$, where $x_1, \ldots, x_n$ are distinct. The rules for typechecking expressions are shown in Figure 2.

We write $\mathcal{M}_{\mathsf{fin}}(X)$ for the set of all finite multisets with elements drawn from X. The (standard) interpretation of base types as sets of values is as follows:

$$\begin{array}{rcl} \mathcal{T}[\![\mathsf{bool}]\!] & = & \mathbb{B} = \{\mathsf{true}, \mathsf{false}\} \\ \mathcal{T}[\![\mathsf{int}]\!] & = & \mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\} \\ \mathcal{T}[\![\tau_1 \times \tau_2]\!] & = & \mathcal{T}[\![\tau_1]\!] \times \mathcal{T}[\![\tau_2]\!] \\ \mathcal{T}[\![\{\tau\}]\!] & = & \mathcal{M}_{\mathsf{fin}}(\mathcal{T}[\![\tau]\!]) \end{array}$$

An environment γ is a function from variables to values. We define the set of environments matching context Γ as $\mathcal{T}[\![\Gamma]\!] = \{\gamma \mid \forall x \in \mathrm{dom}(\Gamma).\ \gamma(x) \in \mathcal{T}[\![\Gamma(x)]\!]\}$.

Figure 3 gives the semantics of queries. Note that we overload notation for pair projection π_i and bag operations such as $\cup$ and $\bigcup$; also, if S is a bag of integers, then $\sum S$ is the sum of their values (taking $\sum \emptyset = 0$). It is straightforward to show that

Lemma 2.1. If $\Gamma \vdash e : \tau$ then $\mathcal{E}[\![e]\!] : \mathcal{T}[\![\Gamma]\!] \rightarrow \mathcal{T}[\![\tau]\!]$.

Remark 2.1. As discussed in previous work [Buneman et al., 1995], the NRC can express a wide variety of queries including ordinary relational queries, nested subqueries, and grouping and aggregation queries. The core NRC excludes a number of convenient features such as records with named fields and comprehensions. However, these features can be viewed as syntactic sugar

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \quad \frac{i \in \mathbb{Z}}{\Gamma \vdash i : \mathsf{int}} \quad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}$$

$$\frac{\Gamma \vdash e : \{\mathsf{int}\}}{\Gamma \vdash \mathsf{sum}(e) : \mathsf{int}} \quad \frac{b \in \mathbb{B}}{\Gamma \vdash b : \mathsf{bool}} \quad \frac{\Gamma \vdash e_0 : \mathsf{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau} \quad \frac{\Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash \neg e : \mathsf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1 \wedge e_2 : \mathsf{bool}} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i(e) : \tau_i}\ (i \in \{1,2\})$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \approx e_2 : \mathsf{bool}} \quad \frac{}{\Gamma \vdash \emptyset : \{\tau\}} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{e\} : \{\tau\}} \quad \frac{\Gamma \vdash e_1 : \{\tau\} \quad \Gamma \vdash e_2 : \{\tau\}}{\Gamma \vdash e_1 \cup e_2 : \{\tau\}}$$

$$\frac{\Gamma \vdash e_1 : \{\tau\} \quad \Gamma \vdash e_2 : \{\tau\}}{\Gamma \vdash e_1 - e_2 : \{\tau\}} \quad \frac{\Gamma \vdash e_1 : \{\tau_1\} \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \{e_2 \mid x \in e_1\} : \{\tau_2\}} \quad \frac{\Gamma \vdash e : \{\{\tau\}\}}{\Gamma \vdash \bigcup e : \{\tau\}}$$

Fig. 2. Well-formed query expressions

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{E}[\![x]\!]\gamma & = & \gamma(x) & \mathcal{E}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\gamma & = & \mathcal{E}[\![e_2]\!]\gamma[x \mapsto \mathcal{E}[\![e_1]\!]\gamma] \\
\mathcal{E}[\![i]\!]\gamma & = & i & \mathcal{E}[\![e_1 + e_2]\!]\gamma & = & \mathcal{E}[\![e_1]\!]\gamma + \mathcal{E}[\![e_2]\!]\gamma \\
\mathcal{E}[\![\mathsf{sum}(e)]\!]\gamma & = & \sum \mathcal{E}[\![e]\!]\gamma & \mathcal{E}[\![b]\!]\gamma & = & b \\
\mathcal{E}[\![\neg e]\!]\gamma & = & \neg \mathcal{E}[\![e]\!]\gamma & \mathcal{E}[\![e_1 \wedge e_2]\!]\gamma & = & \mathcal{E}[\![e_1]\!]\gamma \wedge \mathcal{E}[\![e_2]\!]\gamma \\
\mathcal{E}[\![(e_1, e_2)]\!]\gamma & = & (\mathcal{E}[\![e_1]\!]\gamma, \mathcal{E}[\![e_2]\!]\gamma) & \mathcal{E}[\![\pi_i(e)]\!]\gamma & = & \pi_i(\mathcal{E}[\![e]\!]\gamma) \quad (i \in \{1,2\}) \\
\mathcal{E}[\![\emptyset]\!]\gamma & = & \emptyset & \mathcal{E}[\![\{e\}]\!]\gamma & = & \{\mathcal{E}[\![e]\!]\gamma\} \\
\mathcal{E}[\![e_1 \cup e_2]\!]\gamma & = & \mathcal{E}[\![e_1]\!]\gamma \cup \mathcal{E}[\![e_2]\!]\gamma & \mathcal{E}[\![e_1 - e_2]\!]\gamma & = & \mathcal{E}[\![e_1]\!]\gamma - \mathcal{E}[\![e_2]\!]\gamma \\
\mathcal{E}[\![\bigcup e]\!]\gamma & = & \bigcup \mathcal{E}[\![e]\!]\gamma & \mathcal{E}[\![\{e \mid x \in e_0\}]\!]\gamma & = & \{\mathcal{E}[\![e]\!]\gamma[x \mapsto v] \mid v \in \mathcal{E}[\![e_0]\!]\gamma\}
\end{array}$$

$$\mathcal{E}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\gamma = \begin{cases} \mathcal{E}[\![e_1]\!]\gamma & \text{if } \mathcal{E}[\![e_0]\!]\gamma = \mathsf{true} \\ \mathcal{E}[\![e_2]\!]\gamma & \text{if } \mathcal{E}[\![e_0]\!]\gamma = \mathsf{false} \end{cases}$$

$$\mathcal{E}[\![e_1 \approx e_2]\!]\gamma = \begin{cases} \mathsf{true} & \text{if } \mathcal{E}[\![e_1]\!]\gamma = \mathcal{E}[\![e_2]\!]\gamma \\ \mathsf{false} & \text{if } \mathcal{E}[\![e_1]\!]\gamma \neq \mathcal{E}[\![e_2]\!]\gamma \end{cases}$$

Fig. 3. Semantics of query expressions

$$\begin{array}{rcl}
\Pi_A(R) & = & \{x.A \mid x \in R\} \\
\sigma_{A=B}(R) & = & \bigcup \{\mathsf{if}\ x.A = x.B\ \mathsf{then}\ \{x\}\ \mathsf{else}\ \emptyset \mid x \in R\} \\
R \times S & = & \{(A : x.A, B : x.B, C : y.C, D : y.D, E : y.E) \mid x \in R, y \in S\} \\
\Pi_{BE}(\sigma_{A=D}(R \times S)) & = & \{\mathsf{if}\ x.A = y.D\ \mathsf{then}\ \{(B : x.B, E : y.E)\}\ \mathsf{else}\ \emptyset \mid x \in R, y \in S\} \\
R \cup \rho_{A/C,B/D}(\Pi_{CD}(S)) & = & R \cup \{(A : y.C, B : y.D) \mid y \in S\} \\
R - \rho_{A/D,B/E}(\Pi_{DE}(S)) & = & R - \{(A : y.D, B : y.E) \mid y \in S\} \\
\mathsf{sum}(\Pi_A(R)) & = & \mathsf{sum}\{x.A \mid x \in R\} \\
\mathsf{count}(R) & = & \mathsf{sum}\{1 \mid x \in R\}
\end{array}$$

Fig. 4. Example queries

for core NRC expressions. In particular, we use abbreviations such as:

$$\begin{array}{rcl}\{e \mid x_1 \in e_1, x_2 \in e_2\} & = & \bigcup\{\{e \mid x_2 \in e_2\} \mid x_1 \in e_1\} \\ \{e \mid x \in e_0, C\} & = & \bigcup\{\mathsf{if}\ C\ \mathsf{then}\ \{e\}\ \mathsf{else}\ \emptyset \mid x \in e_0\} \\ \{e \mid (x_1, x_2) \in e_0\} & = & \{\mathsf{let}\ x_1 = \pi_1(x), x_2 = \pi_2(x)\ \mathsf{in}\ e \mid x \in e_0\}\end{array}$$

Additional base types, primitive functions and relations such as real, $/ : \mathsf{real} \times \mathsf{real} \rightarrow \mathsf{real}$, and $\mathsf{average} : \{\mathsf{real}\} \rightarrow \mathsf{real}$ can also be added without difficulty. For example, using more readable named records, comprehensions, and pattern-matching, the SQL query fromFigure 1(b) can be defined as

$$\begin{array}{l}\mathsf{let}\ X = \{(r.Name, p.MW) \mid r \in R, er \in ER, p \in P, er.RID = r.ID, p.ID = er.PID\}\ \mathsf{in} \\ \{(n, \mathsf{average}\{mw \mid (n', mw) \in X, n = n'\}) \mid (n, \_) \in X\}\end{array}$$

Additional examples are shown in Figure 4.

We do not consider other features of SQL such as operator overloading or incomplete information (NULL values).

3. Annotations, Provenance and Dependence

We wish to define *dependency provenance* as information relating each part of the output of a query to a set of parts of the input on which the output part depends. Collection types such as sets and bags are unordered and lack a natural way to address parts of values, so we must introduce one. One technique (familiar from many program analyses [Nielson et al., 2005] as well as other work on provenance [Buneman et al., 2008b, Wang and Madnick, 1990]) is to enrich the data model with *annotations* that can be used to refer to parts of the value. In practice, the annotations might consist of explicit paths or addresses pointing into a particular representation of a part of the data, analogous to filenames and line number references in compiler error messages, but for our purposes, it is preferable to leave the structure of annotations abstract; we therefore consider annotations to be sets of *colors*, or elements of some abstract data type Color.

We can then infer provenance information from functions on annotated values by observing how such functions propagate annotations; conversely, we can define provenance-tracking semantics by enriching ordinary functions with annotation-propagation behavior. However, for any ordinary function, there are many corresponding annotation-propagating functions so the question arises of how to choose among them.

We consider two natural constraints on the annotated functions we will consider. First, if we ignore annotations, the behavior of an annotated function should correspond to that of an ordinary function. Second, the behavior of the annotated functions should treat the annotations abstractly, so that we may view the colors as locations. We show that both properties follow from a single condition called *color-invariance*.

We next define dependency-correctness, a property characterizing annotated functions whose annotations safely over-approximate the dependency behavior of some ordinary function. Such annotations can be used to compute a natural notion of “data slices”, by highlighting those parts of the input on which a given part of the output *may* depend. It is clearly desirable to produce slices that are as small as possible. Unfortunately, minimal slices turn out not to be computable since it is undecidable whether the annotations in the output of a dependency-correct function are minimal. In the next sections, we will show how to calculate approximate dynamic and static dependency information for NRC queries.

We define *annotated values (a-values)* v, *raw values (r-values)* w, and *multisets of annotated values* V as follows:

$$v \quad ::= \quad w^{\Phi} \qquad w ::= i \mid b \mid (v_1, v_2) \mid V \qquad V ::= \{v_1, \ldots, v_n\}$$

Annotations are *sets* $\Phi \subseteq \mathsf{Color}$ of values from some atomic data type Color, called *colors*. We often omit set brackets in the annotations, for example writing $w^{a,b,c}$ instead of $w^{\{a,b,c\}}$ and w instead of $w^{\emptyset}$. An a-value v is said to be *distinctly colored* if every part of it is colored with a singleton set $\{a\}$ and no color c is used more than once in v.

For each type τ, we define the set $\mathcal{A}[\![\tau]\!]$ of annotated values of type τ as follows:

$$\begin{array}{rcl} \mathcal{A}[\![\mathsf{bool}]\!] & = & \{b^{\Phi} \mid b \in \mathbb{B}\} \\ \mathcal{A}[\![\mathsf{int}]\!] & = & \{i^{\Phi} \mid i \in \mathbb{Z}\} \\ \mathcal{A}[\![\tau_1 \times \tau_2]\!] & = & \{(v_1, v_2)^{\Phi} \mid v_1 \in \mathcal{A}[\![\tau_1]\!], v_2 \in \mathcal{A}[\![\tau_2]\!]\} \\ \mathcal{A}[\![\{\tau\}]\!] & = & \{V^{\Phi} \mid \forall v \in V. v \in \mathcal{A}[\![\tau]\!]\} \end{array}$$

Annotated environments $\widehat{\gamma}$ map variables to annotated values. We define the set of annotated environments matching context Γ as $\mathcal{A}[\![\Gamma]\!] = \{\widehat{\gamma} \mid \forall x \in \mathrm{dom}(\Gamma). \widehat{\gamma}(x) \in \mathcal{A}[\![\Gamma(x)]\!]\}$.

We define an *erasure function* $|{-}|$, mapping a-values to ordinary values (and, abusing notation, also mapping r-values to ordinary values), and an *annotation extraction function* $\| - \|$ which extracts the set of all colors mentioned anywhere in an a-value or r-value, as follows:

$$\begin{array}{rclcrcl} |i| & = & i & \qquad & \|i\| & = & \emptyset \\ |b| & = & b & & \|b\| & = & \emptyset \\ |(v_1, v_2)| & = & (|v_1|, |v_2|) & & \|(v_1, v_2)\| & = & \|v_1\| \cup \|v_2\| \\ |\{V\}| & = & \{|v| \mid v \in V\} & & \|\{V\}\| & = & \bigcup\{\|v\| \mid v \in V\} \\ |w^{\Phi}| & = & |w| & & \|w^{\Phi}\| & = & \Phi \cup \|w\| \end{array}$$

Two a-values are said to be *compatible* (written $v \cong v'$) if $|v| = |v'|$; also, an a-value v is said to *enrich* an ordinary value v' (written $v \gtrsim v'$) provided $|v| = v'$.

We now consider annotated functions (a-functions) $F : \mathcal{A}[\![\tau]\!] \to \mathcal{A}[\![\tau']\!]$ on a-values. We say that a-function F enriches an ordinary function $f : \mathcal{T}[\![\tau]\!] \to \mathcal{T}[\![\tau']\!]$ (written $F \gtrsim f$), provided that $\forall v \in \mathcal{A}[\![\tau]\!]. f(|v|) = |F(v)|$. We will also consider annotated functions $F : \mathcal{A}[\![\Gamma]\!] \to \mathcal{A}[\![\tau]\!]$ mapping annotated environments to values. We say that an a-function F enriches an ordinary function $f : \mathcal{T}[\![\Gamma]\!] \to \mathcal{T}[\![\tau]\!]$ (again written $F \gtrsim f$), provided that $\forall \gamma \in \mathcal{A}[\![\Gamma]\!]. f(|\gamma|) = |F(\gamma)|$.

3.1. *Color-invariance*

Clearly, many exotic a-functions exist that are not enrichments of any ordinary function. For example, consider

$$F(i^{\Phi}) = \left\{ \begin{array}{ll} 1^{\Phi} & (\Phi = \emptyset) \\ 0^{\Phi} & (\Phi \neq \emptyset) \end{array} \right.$$

Here, F tests whether its annotation is empty or not, and there is no ordinary function $f : \mathcal{T}[\![\mathsf{int}]\!] \to \mathcal{T}[\![\mathsf{int}]\!]$ such that $\forall v. |F(v)| = f(|v|)$. In the rest of this article we will restrict attention to a-functions F that are enrichments of ordinary functions.

In fact, we will restrict attention still further to a-functions whose behavior on colors is also abstract enough to be consistent with an interpretation of colors in the input as addresses for parts of the input. For example, consider $G, H : \mathcal{A}[\![\mathsf{int}]\!] \to \mathcal{A}[\![\mathsf{int}]\!]$ having the following behavior:

$$\begin{array}{rcl} G(i^{\Phi}) & = & \left\{ \begin{array}{ll} i^{\emptyset} & (\Phi = \emptyset) \\ i^{\{c\}} & (\Phi \neq \emptyset) \end{array} \right. \\ H(i^{\Phi}) & = & i^{\Phi - \{c\}} \end{array}$$

where in both cases c is some fixed color. Both functions are enrichments of the ordinary identity function on integers, $\lambda i.i$, but both perform nontrivial computations on the annotations. If we wish to interpret the colors on these functions as representing sets of locations, then we want to exclude from consideration functions like G whose behavior depends on the size of the annotation set or functions like H whose behavior depends on a specific color.

By analogy with generic queries in relational databases [Abiteboul et al., 1995], such a-functions ought to behave in a way that is insensitive to the particular choice of colors. Moreover, since a-values are annotated by sets of colors, the a-functions also ought to be insensitive to properties of the annotations such as nonemptiness or equality. In particular, we expect that the behavior of an a-function is determined by its behavior on distinctly-colored inputs.

To make this precise, we first need to define some auxiliary concepts.

Definition 3.1. An a-value v is *distinctly-colored* provided every subexpression w^{Φ} we have $\Phi = \{c\}$ for some color c, and no two subexpressions occurring in v have the same color.

Example 3.1. For example, $v = \{(1^a, 1^b)^c\}^d$ is distinctly-colored, while $v' = \{(1^a, 1^a)^c\}^d$ is not, because the color a is re-used in two different subexpression occurrences of 1^a.

A *color substitution* is a function $\alpha : \mathsf{color} \to \{\mathsf{color}\}$ mapping colors to sets of colors. We can lift color substitutions to act on arbitrary a-values as follows:

$$\begin{array}{rcl} \alpha(b) & = & b \\ \alpha(i) & = & i \\ \alpha(v_1, v_2) & = & (\alpha(v_1), \alpha(v_2)) \\ \alpha(V) & = & \{\alpha(v) \mid v \in V\} \\ \alpha(w^{\Phi}) & = & (\alpha(w))^{\alpha[\Phi]} \end{array}$$

where $\alpha[\Phi] = \bigcup\{\alpha(c) \mid c \in \Phi\}$. Note that for any $v \in \mathcal{A}[\![\tau]\!]$, we have $\alpha(v) \in \mathcal{A}[\![\tau]\!]$; we sometimes write α^{τ} to indicate the restriction of α to $\mathcal{A}[\![\tau]\!]$.

Example 3.2. Continuing the previous example, consider the color substitution defined by $\alpha(a) = \{b, c\}$ and $\alpha(x) = \{x\}$ for $x \neq a$. Applying this substitution to v yields $\{(1^{a,b}, 1^b)^c\}^d$. Applying to $\{1^a, 2^{a,d}\}^c$ yields $\{1^{b,c}, 2^{b,c,d}\}^c$.

We note some useful properties relating distinctly-colored values, color-substitution and the erasure and color-support functions; these are easy to prove by induction.

Lemma 3.1. Suppose $\alpha : \mathsf{color} \to \{\mathsf{color}\}$. Then (1) $|\alpha(v)| = |v|$ and (2) $\|\alpha(v)\| = \alpha[\|v\|]$.

Lemma 3.2. Suppose v is an a-value. Then there exists a distinctly-colored $v_0 \cong v$ and a color substitution α_0 such that $\alpha_0(v_0) = v$.

Accordingly, for each ordinary value v fix a distinctly-annotated $\mathsf{dc}(v)$; moreover, for each a-value v fix a color substitution α_v such that $\alpha_v(\mathsf{dc}|v|) = v$.

We now define a property called *color-invariance*, by analogy with the *color-propagation* studied in [Buneman et al., 2008b] for annotations consisting of single colors. Color-invariance is defined as follows.

Definition 3.2 (Color-invariance). An a-function $F : \mathcal{A}[\![\tau_1]\!] \to \mathcal{A}[\![\tau_2]\!]$ is called *color-invariant* if whenever $\alpha : \mathsf{color} \to \{\mathsf{color}\}$ then we have $\alpha^{\tau_2}(F(v)) = F(\alpha^{\tau_1}(v))$.

As noted above, color-invariance has two important consequences. First, the behavior of a color-invariant function is determined by its behavior on distinctly-colored inputs. Second, color-invariant functions are always enrichments of ordinary functions.

Proposition 3.1. If $F, G : \mathcal{A}[\![\tau_1]\!] \to \mathcal{A}[\![\tau_2]\!]$ is color-invariant then the following are equivalent:

1 $F = G$

2 $F(v) = G(v)$ for every *distinctly-colored* $v \in \mathcal{A}[\![\tau_1]\!]$.

3 $F(\mathsf{dc}(v)) = G(\mathsf{dc}(v))$ for every ordinary value $v \in \mathcal{T}[\![\tau_1]\!]$.

Proof. The implications $(1) \Rightarrow (2) \Rightarrow (3)$ are trivial. We show (3) implies (1). Let $v \in \mathcal{A}[\![\tau_1]\!]$ be given. Then $v = \alpha_v(\mathsf{dc}(|v|))$, so to prove $F = G$, it suffices to show:

$$F(v) = F(\alpha_v(\mathsf{dc}(|v|))) = \alpha_v(F(\mathsf{dc}(|v|))) = \alpha_v(G(\mathsf{dc}(|v|))) = G(\alpha_v(\mathsf{dc}(|v|))) = G(v)$$

Proposition 3.2. If $F : \mathcal{A}[\![\tau_1]\!] \to \mathcal{A}[\![\tau_2]\!]$ is color-invariant then $F \gtrsim f$ where $f(v) = |F(\mathsf{dc}(v))|$.

Proof. Let $v \in \mathcal{A}[\![\tau_1]\!]$ be given. Then to prove $F \gtrsim f$, observe:

$$f(|v|) = |F(\mathsf{dc}(|v|))| = |\alpha_v(F(\mathsf{dc}(|v|)))| = |F(\alpha_v(\mathsf{dc}(|v|)))| = |F(v)|$$

We write $|F|$ for f provided $F \gtrsim f$; clearly, f is unique when it exists, and $|F|$ exists for any color-invariant F.

3.2. *Dependency-correctness*

We now turn to the problem of characterizing a-functions whose annotation behavior captures a form of dependency information. Intuitively, an a-function F is dependency-correct if its output annotations tell us how changes to parts of the input may affect parts of the output. First, we need to capture the intuitive notion of changing a specific part of a value.

Definition 3.3 (Equal except at c). Two a-values v_1, v_2 are *equal except at c* ($v_1 \equiv_c v_2$) provided that they have the same structure except possibly at subterms labeled with the color c; this relation is defined as follows:

$$\frac{d \in \mathbb{B} \cup \mathbb{Z}}{d \equiv_c d} \quad \frac{v_1 \equiv_c v_1' \quad v_2 \equiv_c v_2'}{(v_1, v_2) \equiv_c (v_1', v_2')} \quad \frac{v_1 \equiv_c v_1' \quad \cdots \quad v_n \equiv_c v_n'}{\{v_1, \ldots, v_n\} \equiv_c \{v_1', \ldots, v_n'\}} \quad \frac{w_1 \equiv_c w_2}{w_1^{\Phi} \equiv_c w_2^{\Phi}} \quad \frac{c \in \Phi_1 \cap \Phi_2}{w_1^{\Phi_1} \equiv_c w_2^{\Phi_2}}$$

Furthermore, we say that two annotated environments $\widehat{\gamma}, \widehat{\gamma}'$ are equal except at a (written $\widehat{\gamma} \equiv_a \widehat{\gamma}'$) if their domains are compatible ($\mathrm{dom}(\widehat{\gamma}) = \mathrm{dom}(\widehat{\gamma}')$) and they are pointwise equal except at a, that is, for each $x \in \mathrm{dom}(\widehat{\gamma})$, we have $\widehat{\gamma}(x) \equiv_a \widehat{\gamma}'(x)$.

Remark 3.1. For distinctly-colored values, a color serves as an address uniquely identifying a subterm. Thus, $\equiv_c$ relates a distinctly-colored value to a value which can be obtained by modifying the subterm located at c; that is, if we write v_1 as $C[v_1']$ where C is a context and v_1' is the subterm labeled with c in v_1, and $v_1 \equiv_c v_2$, then $v_2 = C[v_2']$ for some subterm v_2' labeled with c. Note that v_2' and v_2 need not be distinctly colored, and that $\equiv_c$ makes sense for arbitrary a-values, not just distinctly colored ones.

Example 3.3. Consider the two a-environments:

$$\begin{array}{rcl} \widehat{\gamma} & = & (\mathrm{R} : \{(1^{c_1}, 3^{c_2}, 5^{c_3})^{b_1}, \ldots\}^a, \mathrm{S} : \cdots) \\ \widehat{\gamma}' & = & (\mathrm{R} : \{(2^{c_1}, 3^{c_2}, 5^{c_3})^{b_1}, \ldots\}^a, \mathrm{S} : \cdots) \end{array}$$

We have $\widehat{\gamma} \equiv_a \widehat{\gamma}'$, $\widehat{\gamma} \equiv_{b_1} \widehat{\gamma}'$, and $\widehat{\gamma} \equiv_{c_1} \widehat{\gamma}'$, assuming that the elided portions are identical.

Definition 3.4 (Dependency-correctness). An a-function $F : \mathcal{A}[\![\Gamma]\!] \to \mathcal{A}[\![\tau]\!]$ is *dependency-correct* if for any $c \in \mathsf{Color}$ and $\widehat{\gamma}, \widehat{\gamma}' \in \mathcal{A}[\![\Gamma]\!]$ satisfying $\widehat{\gamma} \equiv_c \widehat{\gamma}'$, we have $F(\widehat{\gamma}) \equiv_c F(\widehat{\gamma}')$.

Example 3.4. Recall $\widehat{\gamma}, \widehat{\gamma}'$ as in the previous example. Suppose F is dependency-correct and

$$F(\widehat{\gamma}) = \{(1^{c_1}, 3^{c_2}, 5^{c_3})^{b_1}\}^a \ .$$

Since $\widehat{\gamma} \equiv_{c_1} \widehat{\gamma}'$, we know that $F(\widehat{\gamma}) \equiv_{c_1} F(\widehat{\gamma}')$ so we can see that $F(\widehat{\gamma}')$ must be of the form

$$\{(x^{c_1}, 3^{c_2}, 5^{c_3})^{b_1}\}^a$$

for some $x \in \mathbb{Z}$. We do *not* necessarily know that x must be 2; this is not captured by dependency-correctness.

Remark 3.2. Dependency-correctness tells us that for any c, we must have $F(\widehat{\gamma}) = C[v_1, \ldots, v_n]$ and $F(\widehat{\gamma}') = C[v_1', \ldots, v_n']$, where $C[-, \ldots, -]$ is a context not mentioning c and $v_1, \ldots, v_n$, $v_1', \ldots, v_n'$ are labeled with c. Thus, F's annotations tell us which parts of the output (i.e., $v_1, \ldots, v_n$) *may* change if the input is changed at c. Dually, they also tell us what part of the output (i.e., $C[-, \ldots, -]$) *cannot* be changed by changing the input at c.

We can consider the parts of the output labeled with c to be a *forward slice* of the input at c; it shows all of the parts of the output that may depend on c. Conversely, suppose the output is of the form $C'[w^{\Phi}]$. Then we can define a *backward slice* corresponding to this part of the output by factoring $\widehat{\gamma}$ into $C[v_1, \ldots, v_n]$ where C is as small as possible subject to the constraint that $\Phi \cap \|v_i\| = \emptyset$ for each i. This context $C[-, \ldots, -]$ identifies all of the parts of the input on which a given part of the output may depend.

Of course, dependency-correctness does not uniquely characterize the annotation behavior of a given F. It is possible for the annotations to be dependency-correct but inaccurate. For example we can always trivially annotate each part of the output with every color appearing in the input. This, of course, tells us nothing about the function's behavior. In general, the fewer the annotations present in the output of a dependency-correct F, the more they tell us about F's behavior. We therefore consider a function F to be *minimally annotated* if no annotations can be removed from F's output for any v without damaging correctness.

Example 3.5. For example, consider the ordinary function

$$f(x,y) = \begin{cases} y & : x = 0 \\ x+1 & : x \neq 0 \end{cases} .$$

Then the function

$$F(x^a, y^b) = \begin{cases} y^{a,b} & : x = 0 \\ (x+1)^{a,b} & : x \neq 0 \end{cases} .$$

is dependency-correct: trivially so, since it always propagates all annotations from the input to each part of the output. Conversely,

$$G(x^a, y^b) = \begin{cases} y^{a,b} & : x = 0 \\ (x+1)^{a} & : x \neq 0 \end{cases} .$$

is dependency-correct and minimally annotated. To see that G is dependency-correct, note that if we evaluate $G(x,y)$ on $x \neq 0$ then changing only the value of y (annotated by b) can never change the result. To see that G is minimally annotated, it suffices to check that removing any of the annotations breaks dependency-correctness.

We say that a query e is *constant* if $[\![e]\!]\gamma = v$ for some v and every suitable γ. Clearly, a query is constant if and only if it has a dependency-correct enrichment which annotates each part of the result with $\emptyset$.

Proposition 3.3. It is undecidable whether a Boolean NRC query is constant.

Proof. Recall that query equivalence is undecidable for the relational calculus [Abiteboul et al., 1995]; for NRC, equivalence is undecidable for queries $e(x), e'(x)$ over a single variable x. Given two such queries, consider the expression $\widehat{e} = e(x) \approx e'(x) \vee y$ (definable as $\neg(\neg(e(x) \approx e'(x)) \wedge \neg y))$, where y is a fresh variable distinct from x. The result of this expression cannot be false everywhere since the disjunction is true for $y = \mathsf{true}$, so $\widehat{e}$ is constant iff $[\![\widehat{e}]\!]\gamma = \mathsf{true}$ for every γ iff $e \equiv e'$. □

Clearly, an annotation is needed on the result of a Boolean query if and only if the query is not a constant, so finding minimal annotations (or minimal slices) is undecidable. As a result, we cannot expect to be able to compute minimal dependency-correct annotations. It is important to note, though, that dependency-tracking remains hard even if we consider sublanguages for which equivalence is decidable. For example, if we just consider Boolean expressions, finding minimal correct dependency information is also intractable, by an easy reduction from the validity problem. These observations motivate considering approximation techniques, such as those in the next two sections.

Remark 3.3 (Uniqueness of minimum annotations). We have shown that annotation minimality is undecidable. However there is another interesting question that we have not answered: specifically, given a function f, is there a *unique* minimally-annotated function $F \gtrsim f$? To show this, one strategy could be to define a meet (greatest lower bound) operation $v \sqcap w$ on compatible annotated values, lift this to compatible annotated functions $(F \sqcap G)(x) = F(x) \sqcap G(x)$, show

$$
\begin{array}{rcl@{\qquad}rcl}
(w^{\Phi})^{+\Phi_0} & = & w^{\Phi\cup\Phi_0} & (i_1^{\Phi_1}) \mathbin{\widehat{+}} (i_2^{\Phi_2}) & = & (i_1 + i_2)^{\Phi_1\cup\Phi_2} \\
\widehat{\neg}\,(b^{\Phi}) & = & (\neg b)^{\Phi} & (b_1^{\Phi_1}) \mathbin{\widehat{\wedge}} (b_2^{\Phi_2}) & = & (b_1 \wedge b_2)^{\Phi_1\cup\Phi_2} \\
\widehat{\pi}_i((v_1, v_2)^{\Phi}) & = & v_i^{+\Phi} & (w_1^{\Phi_1}) \mathbin{\widehat{\cup}} (w_2^{\Phi_2}) & = & (w_1 \cup w_2)^{\Phi_1\cup\Phi_2} \\
\widehat{\mathsf{cond}}(\mathsf{true}^{\Phi}, v_1, v_2) & = & v_1^{+\Phi} & \widehat{\mathsf{cond}}(\mathsf{false}^{\Phi}, v_1, v_2) & = & v_2^{+\Phi}
\end{array}
$$

$$
\begin{array}{rcl}
\widehat{\sum}(\{v_1, \ldots, v_n\}^{\Phi}) & = & (v_1 \mathbin{\widehat{+}} \cdots \mathbin{\widehat{+}} v_n)^{+\Phi} \\
\widehat{\bigcup}\{v_1, \ldots, v_n\}^{\Phi} & = & (v_1 \mathbin{\widehat{\cup}} \cdots \mathbin{\widehat{\cup}} v_n)^{+\Phi} \\
\{v(x) \mid x \mathrel{\widehat{\in}} w^{\Phi}\} & = & \{v(x) \mid x \in w\}^{\Phi} \\
(w_1^{\Phi_1}) \mathbin{\widehat{-}} (w_2^{\Phi_2}) & = & \{v \in w_1 \mid |v| \notin |w_2|\}^{\Phi_1\cup\|w_1\|\cup\Phi_2\cup\|w_2\|} \\
v_1 \mathrel{\widehat{\approx}} v_2 & = & \begin{cases} \mathsf{true}^{\|v_1\|\cup\|v_2\|} & |v_1| = |v_2| \\ \mathsf{false}^{\|v_1\|\cup\|v_2\|} & |v_1| \neq |v_2| \end{cases}
\end{array}
$$

Fig. 5. Auxiliary annotation-propagating operations

$$
\begin{array}{rcl@{\qquad}rcl}
\mathcal{P}\llbracket x\rrbracket\widehat{\gamma} & = & \widehat{\gamma}(x) & \mathcal{P}\llbracket \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\rrbracket\widehat{\gamma} & = & \mathcal{P}\llbracket e_2\rrbracket(\widehat{\gamma}[x \mapsto \mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma}]) \\
\mathcal{P}\llbracket i\rrbracket\widehat{\gamma} & = & i^{\emptyset} & \mathcal{P}\llbracket e_1 + e_2\rrbracket\widehat{\gamma} & = & (\mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma}) \mathbin{\widehat{+}} (\mathcal{P}\llbracket e_2\rrbracket\widehat{\gamma}) \\
\mathcal{P}\llbracket \mathsf{sum}(e)\rrbracket\widehat{\gamma} & = & \widehat{\sum}(\mathcal{P}\llbracket e\rrbracket\widehat{\gamma}) & \mathcal{P}\llbracket b\rrbracket\widehat{\gamma} & = & b^{\emptyset} \\
\mathcal{P}\llbracket \neg e\rrbracket\widehat{\gamma} & = & \widehat{\neg}(\mathcal{P}\llbracket e\rrbracket\widehat{\gamma}) & \mathcal{P}\llbracket e_1 \wedge e_2\rrbracket\widehat{\gamma} & = & (\mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma}) \mathbin{\widehat{\wedge}} (\mathcal{P}\llbracket e_2\rrbracket\widehat{\gamma}) \\
\mathcal{P}\llbracket (e_1, e_2)\rrbracket\widehat{\gamma} & = & (\mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma}, \mathcal{P}\llbracket e_2\rrbracket\widehat{\gamma})^{\emptyset} & \mathcal{P}\llbracket \pi_i(e)\rrbracket\widehat{\gamma} & = & \widehat{\pi}_i(\mathcal{P}\llbracket e\rrbracket\widehat{\gamma}) \quad (i \in \{1, 2\}) \\
\mathcal{P}\llbracket \emptyset\rrbracket\widehat{\gamma} & = & \emptyset^{\emptyset} & \mathcal{P}\llbracket \{e\}\rrbracket\widehat{\gamma} & = & \{\mathcal{P}\llbracket e\rrbracket\widehat{\gamma}\}^{\emptyset} \\
\mathcal{P}\llbracket e_1 \cup e_2\rrbracket\widehat{\gamma} & = & (\mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma}) \mathbin{\widehat{\cup}} (\mathcal{P}\llbracket e_2\rrbracket\widehat{\gamma}) & \mathcal{P}\llbracket e_1 - e_2\rrbracket\widehat{\gamma} & = & (\mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma}) \mathbin{\widehat{-}} (\mathcal{P}\llbracket e_2\rrbracket\widehat{\gamma}) \\
\mathcal{P}\llbracket \bigcup e\rrbracket\widehat{\gamma} & = & \widehat{\bigcup}\,\mathcal{P}\llbracket e\rrbracket\widehat{\gamma} & \mathcal{P}\llbracket \{e \mid x \in e_0\}\rrbracket\widehat{\gamma} & = & \{\mathcal{P}\llbracket e\rrbracket(\widehat{\gamma}[x \mapsto v]) \mid v \mathrel{\widehat{\in}} \mathcal{P}\llbracket e_0\rrbracket\widehat{\gamma})\} \\
\mathcal{P}\llbracket e_1 \approx e_2\rrbracket\widehat{\gamma} & = & (\mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma}) \mathrel{\widehat{\approx}} (\mathcal{P}\llbracket e_2\rrbracket\widehat{\gamma}) & \mathcal{P}\llbracket \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rrbracket\widehat{\gamma} & = & \widehat{\mathsf{cond}}(\mathcal{P}\llbracket e_0\rrbracket\widehat{\gamma}, \mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma}, \mathcal{P}\llbracket e_2\rrbracket\widehat{\gamma})
\end{array}
$$

Fig. 6. Provenance-tracking semantics

that dependency-correctness is preserved by $\sqcap$, and show that the greatest lower bound of the set of all dependency-correct enrichments of f exists and is dependency-correct.

However, actually defining the meet operation on values that preserves dependency-correctness appears nontrivial. For example, it does not work to simply define the meet as the pointwise intersection of corresponding annotations. Indeed, this is not even well-defined since the "pointwise intersection" of $\{1^a, 1^b\}$ with itself could either be $\{1^a, 1^b\}$ or $\{1^{\emptyset}, 1^{\emptyset}\}$. We therefore leave the uniqueness of minimally annotated functions as a conjecture.

4. Dynamic Provenance Tracking

We now consider a *provenance tracking* approach in which we interpret each expression e as a dependency-correct a-function $\mathcal{P}\llbracket e\rrbracket$. The definition of the provenance-tracking semantics is shown in Figure 6. Auxiliary operations are used to define $\mathcal{P}\llbracket -\rrbracket$; these are shown in Figure 5. In particular, note that we define an auxiliary operation $(w^{\Phi})^{+\Psi} = w^{\Phi\cup\Psi}$ that adds Ψ to the top-level annotation of an a-value w^{Φ}.

Many cases involving ordinary programming constructs are self-explanatory. Constants always have empty annotations: nothing in the input can affect them. Built-in functions such as $+, \wedge, \neg$

propagate all annotations on their arguments to the result. For a conditional $\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, the result is obtained by evaluating e_1 or e_2, and combining the top-level annotation of the result with that of e_0. A constructed pair has an empty top-level annotation; in a projection, the top-level annotation of the pair is merged with that of the returned value.

In the case for let-binding, note that we bind x to the annotated result of evaluating e_1, and then evaluate e_2. It is possible for the dependencies involved in constructing x to not be propagated to the result, if x does not happen to be used in evaluating e_2. This safe because query expressions involve neither stateful side-effects nor nontermination, in contrast to most work on information flow and slicing in general-purpose languages. Similarly, dependencies can be discarded in pair projection expressions $\pi_i(e)$ and set comprehensions $\{e_2 \mid x \in e_2\}$, and again this is safe because queries are purely functional and terminating.

The cases involving collection types deserve further explanation. The empty set is a constant, so has an empty top-level annotation. Similarly, a singleton set constructor has an empty annotation. For union, we take the union of the underlying bags (of annotated values) and fuse the top-level annotations. For comprehension, we leave the top-level annotation alone. For flattening $\bigcup e$, we take the lifted union ($\widehat{\bigcup}$) of the elements of e and add the top-level annotation of e. Similarly, $\widehat{\mathsf{sum}}(e)$ uses $\widehat{+}$ to add together the elements of e, fusing their annotations with that of e. For set difference, to ensure dependency correctness, we must conservatively include all of the colors present on either side in the annotation of the top-level expression. Similarly, for equality tests, we must include all of the colors present in either value in the result annotation.

Note that equivalent expressions $e \equiv e'$ need not satisfy $\mathcal{P}[\![e]\!] \equiv \mathcal{P}[\![e']\!]$; for example, $x - x \equiv \emptyset$ but $\mathcal{P}[\![x - x]\!] \not\equiv \emptyset^{\emptyset}$, since if $\widehat{\gamma}(x) = \{1^d\}^c$ then $\mathcal{P}[\![x - x]\!]\widehat{\gamma} = \emptyset^{c,d}$.

Example 4.1. Consider an annotated input environment $\widehat{\gamma}$, shown in Figure 7(a), of schema $R : \{(A : \mathsf{int}, B : \mathsf{int})\}, S : \{(C : \mathsf{int}, D : \mathsf{int}, E : \mathsf{int})\}$ (we again use named-record syntax for readability). Figure 7(b) shows the provenance tracking semantics of the example queries from Figure 4. We write a_{123} as an abbreviation for the set $\{a_1, a_2, a_3\}$, etc. Note that in the count example query, the output depends only on the number of rows in the input and not on the field values; we cannot change the number of elements of a multiset by changing field values.

Example 4.2 (Grouping and aggregation). Consider a query that performs grouping and aggregation, such as

```
SELECT A, SUM(B) FROM R GROUP BY A
```

First, let

$$X = \{(A : x.A, B : \{y.B \mid y \in R, x.A = y.A\}) \mid x \in R\}$$

When run against the environment $\widehat{\gamma}$ in Figure 7(a), we obtain result

$$X = \{(A : 1^{a_1}, B : \{1^{b_1}, 2^{b_2}\}^{a_{123}}), (A : 1^{a_2}, B : \{1^{b_1}, 2^{b_2}\}^{a_{123}}), (A : 2^{a_3}, B : \{3^{b_3}\}^{a_{123}})\}$$

Note that since we consider collections to be multisets, we get two copies of $(1, \{1, 2\})$, one corresponding to a_1 and one corresponding to a_2. Also, since the subqueries computing the B-values inspect the A-values, each of the groups depends on each of the A-values. We can obtain

(a)

$$\begin{aligned}\widehat{\gamma} &= [R := \{(A:1^{a_1}, B:1^{b_1}), (A:1^{a_2}, B:2^{b_2}), (A:2^{a_3}, B:3^{b_3})\},\\ &\quad\ S := \{(C:1^{c_1}, D:2^{d_1}, E:3^{e_1}), (C:1^{c_2}, D:1^{d_2}, E:4^{e_2})\}]\end{aligned}$$

(b)

$$\begin{aligned}
\mathcal{P}[\![\Pi_A(R)]\!]\widehat{\gamma} &= \{(A:1^{a_1}), (A:1^{a_2}), (A:2^{a_3})\}\\
\mathcal{P}[\![\sigma_{A=B}(R)]\!]\widehat{\gamma} &= \{(A:1^{a_1}, B:1^{b_1})\}^{a_{123}, b_{123}}\\
\mathcal{P}[\![R \times S]\!]\widehat{\gamma} &= \{(A:1^{a_1}, B:1^{b_1}, C:1^{c_1}, D:2^{d_1}, E:3^{e_1}),\\
&\quad (A:1^{a_1}, B:1^{b_1}, C:1^{c_2}, D:1^{d_2}, E:4^{e_2}), \ldots\}\\
\mathcal{P}[\![\Pi_{BE}(\sigma_{A=D}(R \times S))]\!]\widehat{\gamma} &= \{(B:1^{b_1}, E:4^{e_2}), (B:2^{b_2}, E:4^{e_2}),\\
&\quad (B:3^{b_3}, E:3^{e_1})\}^{a_{123}, d_{12}}\\
\mathcal{P}[\![R \cup \rho_{A/C,B/D}(\Pi_{CD}(S))]\!]\widehat{\gamma} &= \{(A:1^{a_1}, B:1^{b_1}), (A:1^{a_2}, B:2^{b_2}), (A:2^{a_3}, B:3^{b_3}),\\
&\quad (A:1^{c_1}, B:2^{d_1}), (A:1^{c_2}, B:1^{d_2})\}\\
\mathcal{P}[\![R - \rho_{A/D,B/E}(\Pi_{DE}(S))]\!]\widehat{\gamma} &= \{(A:1^{a_1}, B:1^{b_2}), (A:1^{a_2}, B:2^{b_2})\}^{a_{123}, b_{123}, d_{12}, e_{12}}\\
\mathcal{P}[\![\mathsf{sum}(\Pi_A(R))]\!]\widehat{\gamma} &= 4^{a_1, a_2, a_3}\\
\mathcal{P}[\![\mathsf{count}(R)]\!]\widehat{\gamma} &= 3\\
\mathcal{P}[\![\mathsf{count}(\sigma_{A=B}(R))]\!]\widehat{\gamma} &= 1^{a_{123}, b_{123}}
\end{aligned}$$

Fig. 7. (a) Annotated input environment (b) Examples of provenance tracking

the final result of aggregation by evaluating

$$\begin{aligned}Y &= \{(A: x.A, B: \mathsf{sum}(x.B)) \mid x \in X\}\\ &= \{(A:1^{a_1}, B:3^{a_{123}b_{12}}), (A:1^{a_2}, B:3^{a_{123}b_{12}}), (A:2^{a_3}, B:3^{a_{123}b_3})\}\end{aligned}$$

Remark 4.1. Our approach to handling negation and equality may result in large annotations in some cases. For example, consider $\{1^a, 2^b\}^c - \{1^d, 3^e\}^f$. Changing any of the input locations a, b, c, d, e, f can cause the output to change. For example, changing 1^a to 4^a yields result $\{4, 2\}$, while changing 2^b to 3^b yields result $\emptyset$. Thus, we must include all of the colors in the input in the annotation of the top-level of the result set, since the size of the set can be affected by changes to any of these parts.

Most previous techniques have not attempted to deal with negation. One exception is Cui et al. [2000]'s definition of lineage. In their approach, the lineage of tuple $t \in R - S$ would be the tuple $t \in R$ and all tuples of S. While this is more concise in some cases, it is not dependency-correct by our definition.

On the other hand, our approach can also be more concise than lineage in the presence of negation, because lineage only deals with annotations at the level of records. For example, in $\{1\} - \{\pi_1(x) \mid x \in S\}$, our approach will indicate that the output does not depend on the second components of elements of S, whereas the lineage of each tuple in the result of this query includes all the records in S. This can make a big difference if there are many fields that are never referenced; indeed, some scientific databases have tens or hundreds of fields per record, only a few of which are needed for most queries.

Thus, although our approach to negation does exhibit pathological behavior in some cases, it also provides more useful provenance for other typical queries. In any case all other approaches either ignore negation or also have some pathological behavior. Developing more sophisticated forms of dependence that are better-behaved in the presence of negation is an interesting area for future work.

4.1. *Correctness of dynamic tracking*

In this section, we prove two correctness properties of dynamic tracking. First, we show that if $\Gamma \vdash e : \tau$ then $\mathcal{P}[\![e]\!] : \mathcal{A}[\![\Gamma]\!] \to \mathcal{A}[\![\tau]\!]$ and $\mathcal{P}[\![e]\!] \gtrsim \mathcal{E}[\![e]\!]$, that is, the provenance semantics respects the typing and the ordinary semantics of e. Second, and more importantly, we show that $\mathcal{P}[\![e]\!]$ is dependency-correct. We first establish useful auxiliary properties of the annotation-merging operation $v^{+\Phi}$ and prove that the lifted operations such as $\widehat{+}$ have appropriate types and enrich the corresponding ordinary operations.

Lemma 4.1. Let v be an a-value and Φ an annotation. Then (1) $|v^{+\Phi}| = |v|$ and (2) $\|v^{+\Phi}\| = \|v\| \cup \Phi$.

Lemma 4.2. In the following, assume that v, v_1, v_2 are in the domains of the appropriate functions.

1 $\widehat{+}$: $\mathcal{A}[\![\mathsf{int}]\!] \times \mathcal{A}[\![\mathsf{int}]\!] \to \mathcal{A}[\![\mathsf{int}]\!]$ is color-invariant and $|v_1 \mathbin{\widehat{+}} v_2| = |v_1| + |v_2|$.
2 $\widehat{\sum}$: $\mathcal{A}[\![\{\mathsf{int}\}]\!] \to \mathcal{A}[\![\mathsf{int}]\!]$ is color-invariant and $|\widehat{\sum} v| = \sum |v|$.
3 $\widehat{\neg}$: $\mathcal{A}[\![\mathsf{bool}]\!] \to \mathcal{A}[\![\mathsf{bool}]\!]$ is color-invariant and $|\widehat{\neg} v| = \neg |v|$.
4 $\widehat{\wedge}$: $\mathcal{A}[\![\mathsf{bool}]\!] \times \mathcal{A}[\![\mathsf{bool}]\!] \to \mathcal{A}[\![\mathsf{bool}]\!]$ is color-invariant and $|v_1 \mathbin{\widehat{\wedge}} v_2| = |v_1| \wedge |v_2|$.
5 For any τ_1, τ_2 and $i \in \{1, 2\}$ we have $\widehat{\pi}_i : \mathcal{A}[\![\tau_1 \times \tau_2]\!] \to \mathcal{A}[\![\tau_i]\!]$ is color-invariant and $|\widehat{\pi}_i(v)| = \pi_i(|v|)$.
6 For any τ, we have $\widehat{\approx} : \mathcal{A}[\![\tau]\!] \times \mathcal{A}[\![\tau]\!] \to \mathcal{A}[\![\mathsf{bool}]\!]$ is color-invariant and $|v_1 \mathbin{\widehat{\approx}} v_2| = (|v_1| \approx |v_2|)$.
7 For any τ, we have $\widehat{\mathsf{cond}} : \mathcal{A}[\![\mathsf{bool}]\!] \times \mathcal{A}[\![\tau]\!] \times \mathcal{A}[\![\tau]\!] \to \mathcal{A}[\![\tau]\!]$ is color-invariant and $|\widehat{\mathsf{cond}}(v, v_1, v_2)| =$ $\mathsf{if}\ |v|\ \mathsf{then}\ |v_1|\ \mathsf{else}\ |v_2|$.
8 For any τ, we have $\widehat{\cup} : \mathcal{A}[\![\{\tau\}]\!] \times \mathcal{A}[\![\{\tau\}]\!] \to \mathcal{A}[\![\{\tau\}]\!]$ is color-invariant and $|v_1 \mathbin{\widehat{\cup}} v_2| = |v_1| \cup |v_2|$.
9 For any τ, we have $\widehat{-} : \mathcal{A}[\![\{\tau\}]\!] \times \mathcal{A}[\![\{\tau\}]\!] \to \mathcal{A}[\![\{\tau\}]\!]$ is color-invariant and $|v_1 \mathbin{\widehat{-}} v_2| = |v_1| - |v_2|$.
10 For any τ, we have $\widehat{\bigcup} : \mathcal{A}[\![\{\{\tau\}\}]\!] \to \mathcal{A}[\![\{\tau\}]\!]$ is color-invariant and $|\widehat{\bigcup} v| = \bigcup |v|$.

Proof. Most cases are immediate. The cases for sum ($\sum$) and flattening ($\bigcup$) rely on the cases for binary addition and union.

The second part of the case of difference (9) is slightly involved. We reason as follows.

$$\begin{aligned} |w_1^{\Phi_1} \mathbin{\widehat{-}} w_2^{\Phi_2}| &= |\{v \mid v \in w_1, |v| \notin |w_2|\}^{\Phi_1 \cup \|w_1\| \cup \Phi_2 \cup \|w_2\|}| = \{|v| \mid v \in w_1, |v| \notin |w_2|\} \\ &= \{v \mid v \in |w_1|, v \notin |w_2|\} = |w_1| - |w_2| \end{aligned}$$

Lemma 4.3. If $\Gamma \vdash e : \tau$ then $\mathcal{P}[\![e]\!] : \mathcal{A}[\![\Gamma]\!] \to \mathcal{A}[\![\tau]\!]$ is color-invariant and $\mathcal{P}[\![e]\!] \gtrsim \mathcal{E}[\![e]\!]$.

Proof. Proof is by induction on expressions e (which determine the structure of the typing judgment). Most cases are straightforward, given Lemma 4.2; we show the case of comprehensions.

— Case $e = \{e_2 \mid x \in e_1\}$:

$$\frac{\Gamma \vdash e_1 : \{\tau_1\} \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \{e_2 \mid x \in e_1\} : \{\tau_2\}}$$

First, by induction we have $\mathcal{P}\llbracket e_1 \rrbracket : \mathcal{A}\llbracket \Gamma \rrbracket \to \mathcal{A}\llbracket \{\tau_1\} \rrbracket$. Hence $w^{\Phi} := \mathcal{P}\llbracket e_1 \rrbracket \widehat{\gamma}$ is a set of a-values in $\mathcal{A}\llbracket \tau_1 \rrbracket$. So for each $v \mathrel{\widehat{\in}} \mathcal{P}\llbracket e_1 \rrbracket \widehat{\gamma}$, we have $\widehat{\gamma}' := \widehat{\gamma}[x := v] \in \mathcal{A}\llbracket \Gamma, x{:}\tau_1 \rrbracket$, hence $\mathcal{P}\llbracket e_2 \rrbracket \widehat{\gamma}' \in \mathcal{A}\llbracket \tau_2 \rrbracket$, and so

$$\mathcal{P}\llbracket \{e_2 \mid x \in e_1\} \rrbracket \widehat{\gamma} = \{\mathcal{P}\llbracket e_2 \rrbracket \widehat{\gamma}[x := v] \mid v \in w\}^{\Phi} \in \mathcal{A}\llbracket \{\tau_2\} \rrbracket$$

Furthermore, if $\alpha : \mathsf{color} \to \{\mathsf{color}\}$, then we have

$$\begin{array}{rcl}
\alpha(\mathcal{P}\llbracket \{e_2 \mid x \in e_1\} \rrbracket \widehat{\gamma}) & = & \alpha(\{\mathcal{P}\llbracket e_2 \rrbracket (\widehat{\gamma}[x := v]) \mid v \in w\}^{\Phi}) \\
 & = & \{\alpha(\mathcal{P}\llbracket e_2 \rrbracket (\widehat{\gamma}[x := v])) \mid v \in w\}^{\alpha[\Phi]} \\
 & = & \{\mathcal{P}\llbracket e_2 \rrbracket (\alpha(\widehat{\gamma})[x := \alpha(v)]) \mid v \in w\}^{\alpha[\Phi]} \\
 & = & \{\mathcal{P}\llbracket e_2 \rrbracket (\alpha(\widehat{\gamma})[x := v]) \mid v \in \alpha(w)\}^{\alpha[\Phi]} \\
 & = & \{\mathcal{P}\llbracket e_2 \rrbracket (\alpha(\widehat{\gamma})[x := v]) \mid v \mathrel{\widehat{\in}} \alpha(w)^{\alpha[\Phi]}\} \\
 & = & \{\mathcal{P}\llbracket e_2 \rrbracket (\alpha(\widehat{\gamma})[x := v]) \mid v \mathrel{\widehat{\in}} \alpha(\mathcal{P}\llbracket e_1 \rrbracket \widehat{\gamma})\} \\
 & = & \{\mathcal{P}\llbracket e_2 \rrbracket (\alpha(\widehat{\gamma})[x := v]) \mid v \mathrel{\widehat{\in}} \mathcal{P}\llbracket e_1 \rrbracket \alpha(\widehat{\gamma})\} \\
 & = & \mathcal{P}\llbracket \{e_2 \mid x \in e_1\} \rrbracket \alpha(\widehat{\gamma})
\end{array}$$

where we appeal to the induction hypothesis to show that $\mathcal{P}\llbracket e_1 \rrbracket$ and $\mathcal{P}\llbracket e_2 \rrbracket$ are color-invariant. Hence $\mathcal{P}\llbracket \{e_2 \mid x \in e_1\} \rrbracket$ is color-invariant.

Second, to show that $\mathcal{P}\llbracket \{e_2 \mid x \in e_1\} \rrbracket \gtrsim \mathcal{E}\llbracket \{e_2 \mid x \in e_1\} \rrbracket$, we have:

$$\begin{array}{rcl}
\mathcal{E}\llbracket \{e_2 \mid x \in e_1\} \rrbracket |\widehat{\gamma}| & = & \{\mathcal{E}\llbracket e_2 \rrbracket (|\widehat{\gamma}|[x := v]) \mid v \in \mathcal{E}\llbracket e_1 \rrbracket |\widehat{\gamma}|\} = \{\mathcal{E}\llbracket e_2 \rrbracket (|\widehat{\gamma}|[x := v]) \mid v \in |\mathcal{P}\llbracket e_1 \rrbracket \widehat{\gamma}|\} \\
 & = & \{\mathcal{E}\llbracket e_2 \rrbracket (|\widehat{\gamma}|[x := v]) \mid v \in |w^{\Phi}|\} = \{\mathcal{E}\llbracket e_2 \rrbracket (|\widehat{\gamma}|[x := |v|]) \mid |v| \in |w^{\Phi}|\} \\
 & = & \{\mathcal{E}\llbracket e_2 \rrbracket (|\widehat{\gamma}|[x := |v|]) \mid v \in w\} = \{\mathcal{E}\llbracket e_2 \rrbracket |\widehat{\gamma}[x := v]| \mid v \in w\} \\
 & = & \{|\mathcal{P}\llbracket e_2 \rrbracket (\widehat{\gamma}[x := v])| \mid v \in w\} = |\{\mathcal{P}\llbracket e_2 \rrbracket (\widehat{\gamma}[x := v]) \mid v \in w\}^{\Phi}| \\
 & = & |\{\mathcal{P}\llbracket e_2 \rrbracket (\widehat{\gamma}[x := v]) \mid v \mathrel{\widehat{\in}} w^{\Phi}\}| = |\{\mathcal{P}\llbracket e_2 \rrbracket (\widehat{\gamma}[x := v]) \mid v \mathrel{\widehat{\in}} \mathcal{P}\llbracket e_1 \rrbracket \widehat{\gamma}\}| \\
 & = & |\mathcal{P}\llbracket \{e_2 \mid x \in e_1\} \rrbracket \widehat{\gamma}|
\end{array}$$

□

We now turn to dependency-correctness. Since $\mathcal{P}\llbracket e \rrbracket$ is defined in terms of the special annotation-propagating operations introduced in Figure 5, we need to show that these operations are dependency-correct. We first need to establish properties of $\equiv_a$:

Lemma 4.4.

1 If $v \equiv_a v'$ then $a \in \|v\| \iff a \in \|v'\|$.
2 If $a \notin \|v\|$ and $v \equiv_a v'$ then $v = v'$.
3 If $v_1 \equiv_a v_2$ then $v_1^{+\Phi} \equiv_a v_2^{+\Phi}$.

Proof. The first part is easy to establish by induction on derivations of $\equiv_a$, by noting that $a \in \|v\| \iff a \in \|v'\|$ is equivalent to $\|v\| \cap \{a\} = \|v'\| \cap \{a\}$ and reasoning equationally.

For the second part, note that the rule

$$\frac{w_1 \equiv_a w_2 \quad a \in \Phi_1 \cap \Phi_2}{w_1^{\Phi_1} \equiv_a w_2^{\Phi_2}}$$

can never apply since $a \notin \|w_1^{\Phi_1}\| = \|w_1\| \cup \Phi_1$ implies $a \notin \Phi_1 \cap \Phi_2$. The remaining rules coincide with the rules for annotated value equality.

For the third part observe that both of the rules defining $\equiv_a$ for annotated values are preserved by adding equal sets of annotations to both sides. □

We now state a key lemma which shows that all of the lifted operations are dependency-correct. Many of the arguments are similar. In each case, if we know that the inputs to an operation are $\equiv_a$, we reason by cases on the structure of the derivation of $\equiv_a$. If any of the assumptions $v \equiv_a v'$ hold because $a \in \Phi \cap \Phi'$ for some pair of inputs v, v', then both outputs will also be annotated with a. Otherwise, the inputs must have the same top-level structure, so in each case we have enough information to evaluate the unlifted function and show that the results are still $\equiv_a$.

The proofs for equality and difference operations are slightly different. Both operations are potentially global, that is, changes deep in the input values can affect the top-level structure of the result (trivially for $\approx$, since there is no deep structure in the boolean result). This is, essentially, why we need to include all of the annotations of the inputs in the result of an equality or difference operation. We should point out that this inaccuracy is an area where we believe improvement may be possible, through refining the definition of $\equiv_a$; but this is left for future work.

Lemma 4.5. If $v \equiv_a v'$, $v_1 \equiv_a v_1', v_2 \equiv v_2', \ldots$ then:

1 $v_1 \mathbin{\widehat{+}} v_2 \equiv_a v_1' \mathbin{\widehat{+}} v_2'$
2 $\widehat{\sum} v \equiv_a \widehat{\sum} v'$
3 $\widehat{\neg} v \equiv_a \widehat{\neg} v'$
4 $v_1 \mathbin{\widehat{\wedge}} v_2 \equiv_a v_1' \mathbin{\widehat{\wedge}} v_2'$
5 $\widehat{\pi}_i(v) \equiv_a \widehat{\pi}_i(v')$
6 $v_1 \mathbin{\widehat{\approx}} v_2 \equiv_a v_1' \mathbin{\widehat{\approx}} v_2'$
7 $\widehat{\mathsf{cond}}(v, v_1, v_2) \equiv_a \widehat{\mathsf{cond}}(v', v_1', v_2')$
8 $v_1 \mathbin{\widehat{\cup}} v_2 \equiv_a v_1' \mathbin{\widehat{\cup}} v_2'$
9 $v_1 \mathbin{\widehat{-}} v_2 \equiv_a v_1' \mathbin{\widehat{-}} v_2'$
10 $\widehat{\bigcup} v \equiv_a \widehat{\bigcup} v'$

Proof. For part (1), suppose $v_i = n_i^{\Phi_i}$ and $v_i' = m_i^{\Psi_i}$ for $i \in \{1, 2\}$. There are four cases, depending on the derivations of $n_i \equiv_a m_i$ for $i \in \{1, 2\}$. If both derivations follow because $n_i^{\Phi_i} = m_i^{\Psi_i}$ then $\mathcal{P}[\![e]\!]\widehat{\gamma} = (n_1 + n_2)^{\Phi_1 \cup \Phi_2} = (m_1 + m_2)^{\Psi_1 \cup \Psi_2} = \mathcal{P}[\![e]\!]\widehat{\gamma}'$ so again $\mathcal{P}[\![e]\!]\widehat{\gamma} \equiv_a \mathcal{P}[\![e]\!]\widehat{\gamma}'$. Otherwise one or both of the derivations follows because $a \in \Phi_i \cap \Psi_i$ for $i = 1$ or $i = 2$. Then $a \in (\Phi_1 \cup \Phi_2) \cap (\Psi_1 \cup \Psi_2)$ so again $\mathcal{P}[\![e]\!]\widehat{\gamma} \equiv_a \mathcal{P}[\![e]\!]\widehat{\gamma}'$.

For part (2), there are two cases. If the summed sets are $\equiv_a$ because their top-level annotations mention a, then the results of the sums will also mention a, so we are done. Otherwise, we must have that the summed sets are of equal size and their elements are pairwise matched by $\equiv_a$; hence, we can apply part (1) repeatedly (and then Lemma 4.5) to show that the results are $\equiv_a$.

Parts (3,4) are similar to part (1).

For part (5), suppose $v = (v_1, v_2)^\Phi$ and $v' = (v'_1, v'_2)^{\Phi'}$. Note that

$$\widehat{\pi}_i(v) = \widehat{\pi}_i(v_1, v_2)^\Phi = v_i^{+\Phi}$$

and similarly $\widehat{\pi}_i(v') = (v'_i)^{+\Phi'}$. There are two cases depending on the last step in the derivation of $v \equiv_a v'$. If $a \in \Phi \cap \Phi'$ then we are done since a will be in the top-level annotations of both $v_i^{+\Phi}$ and $(v'_i)^{+\Phi'}$. Otherwise we must have $v_i \equiv_a v'_i$ for $i \in \{1, 2\}$, so again $v_i^{+\Phi} \equiv_a (v'_i)^{+\Phi'}$.

For part (6), there are two cases. If $a \in (\|v_1\| \cup \|v_2\|) \cap (\|v'_1\| \cup \|v'_2\|)$ then we are done. Otherwise by Lemma 4.4, a cannot appear anywhere in v_1, v_2, v'_1, v'_2, so we must have $v_1 = v'_1, v_2 = v'_2$. Hence $(v_1 \approx v_2) = (v'_1 \approx v'_2)$ which implies the two sides are $\equiv_a$ as well.

For part (7), suppose $v = b^\Phi, v' = (b')^{\Phi'}$. If $a \in \Phi \cap \Phi'$ then we are done since both conditionals will have a in their top-level annotation. Otherwise we must have $b = b'$ so $\widehat{\mathsf{cond}}(v, v_1, v_2) = v_i$ and $\widehat{\mathsf{cond}}(v', v'_1, v'_2) = v'_i$, so by induction (and Lemma 4.4) we are done.

For part (8), suppose $v_i = w_i^{\Phi_i}$ and similarly for v'_i. Again if $a \in (\Phi_1 \cup \Phi_2) \cap (\Phi'_1 \cup \Phi'_2)$ then we are done. Otherwise we must have that $w_1 = \{v_{11}, \ldots, v_{1n}\}$, $w'_1 = \{v'_{11}, \ldots, v'_{1n}\}$ where $v_{1i} \equiv_a v'_{1i}$ for each $i \in \{1, \ldots, n\}$, and similarly for w_2, w'_2. Hence the elements of the union of the two multisets can be matched up using the $\equiv_a$ relation, so we can conclude that $w_1 \cup w_2 \equiv_a w'_1 \cup w'_2$ as well. We must also have $\Phi_i = \Phi'_i$ for each $i \in \{1, 2\}$, so we can conclude that

$$v_1 \cup v_2 = (w_1 \cup w_2)^{\Phi_1 \cup \Phi_2} \equiv_a (w'_1 \cup w'_2)^{\Phi'_1 \cup \Phi'_2} = v'_1 \cup v'_2$$

For part (9), the reasoning is similar to part (6).

For part (10), the reasoning is similar to that for part (2), appealing to part (8) once we have expanded to binary unions. □

We conclude the section with the proof of dependency-correctness. It is much simplified by the previous lemma, since many cases now consist only of applying the induction hypothesis and then using dependency-correctness of a lifted operation.

Theorem 4.1. If $\Gamma \vdash e : \tau$ then $\mathcal{P}[\![e]\!]$ is dependency-correct.

Proof. Suppose $\widehat{\gamma} \equiv_a \widehat{\gamma}'$. Again proof is by induction on the structure of expressions/typing derivations. Many cases are immediate using the induction hypothesis and the corresponding parts of Lemma 4.5. We show the remaining cases:

— Case $e = x$:

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}$$

By assumption $\mathcal{P}[\![x]\!]\widehat{\gamma} = \widehat{\gamma}(x) \equiv_a \widehat{\gamma}'(x) = \mathcal{P}[\![x]\!]\widehat{\gamma}'$.

— Case $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

By induction $\mathcal{P}[\![e_1]\!]$ and $\mathcal{P}[\![e_2]\!]$ are dependency-correct. Hence $\mathcal{P}[\![e_1]\!]\widehat{\gamma} \equiv_a \mathcal{P}[\![e_1]\!]\widehat{\gamma}'$, so

$\widehat{\gamma}[x := \mathcal{P}\llbracket e_1 \rrbracket \widehat{\gamma}] \equiv_a \widehat{\gamma}'[x := \mathcal{P}\llbracket e_1 \rrbracket \widehat{\gamma}']$. It then follows by induction that $\mathcal{P}\llbracket e \rrbracket \widehat{\gamma} = \mathcal{P}\llbracket e_2 \rrbracket (\widehat{\gamma}[x := \mathcal{P}\llbracket e_1 \rrbracket \widehat{\gamma}]) \equiv_a \mathcal{P}\llbracket e_2 \rrbracket (\widehat{\gamma}'[x := \mathcal{P}\llbracket e_1 \rrbracket \widehat{\gamma}']) = \mathcal{P}\llbracket e \rrbracket \widehat{\gamma}'$.

- Case $e = (e_1, e_2)$:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

  By induction, $\mathcal{P}\llbracket e_1 \rrbracket$ and $\mathcal{P}\llbracket e_2 \rrbracket$ are dependency-correct, so $v_i = \mathcal{P}\llbracket e_i \rrbracket \widehat{\gamma} \equiv_a \mathcal{P}\llbracket e_i \rrbracket \widehat{\gamma}' = v_i'$ for $i \in \{1, 2\}$. Hence we can immediately derive $(v_1, v_2)^{\emptyset} \equiv_a (v_1', v_2')^{\emptyset}$.

- Case $e = \{e'\}$:

$$\frac{\Gamma \vdash e' : \tau}{\Gamma \vdash \{e'\} : \{\tau\}}$$

  By induction, $\mathcal{P}\llbracket e' \rrbracket$ is dependency-correct, so $v = \mathcal{P}\llbracket e' \rrbracket \widehat{\gamma} \equiv_a \mathcal{P}\llbracket e' \rrbracket \widehat{\gamma}' = v'$. Hence we can immediately derive $\{v\}^{\emptyset} \equiv_a \{v'\}^{\emptyset}$.

- Case $e = \{e_2 \mid x \in e_1\}$:

$$\frac{\Gamma \vdash e_1 : \{\tau_1\} \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \{e_2 \mid x \in e_1\} : \{\tau_2\}}$$

  By induction, $\mathcal{P}\llbracket e_1 \rrbracket$ and $\mathcal{P}\llbracket e_2 \rrbracket$ are dependency-correct. Hence $w_1^{\Phi_1} = \mathcal{P}\llbracket e_1 \rrbracket \widehat{\gamma} \equiv_a \mathcal{P}\llbracket e_1 \rrbracket \widehat{\gamma}' = (w_1')^{\Phi_1'}$. There are two cases. If $a \in \Phi_1 \cap \Phi_1'$ then we are done since $\mathcal{P}\llbracket e \rrbracket \widehat{\gamma}$ and $\mathcal{P}\llbracket e \rrbracket \widehat{\gamma}'$ will both contain top-level annotations a. Otherwise, we must have

$$\frac{\Phi_1 = \Phi_1' \qquad \dfrac{v_{11} \equiv_a v_{11}' \quad \cdots \quad v_{1n} \equiv_a v_{1n}'}{w_1 \equiv_a w_1'}}{w_1^{\Phi_1} \equiv_a (w_1')^{\Phi_1'}}$$

  where $w_1 = \{v_{11}, \ldots, v_{1n}\}$ and similarly for w_1'. Thus, for each $i \in \{1, \ldots, n\}$, we have $\widehat{\gamma}[x := v_{1i}] \equiv_a \widehat{\gamma}'[x := v_{1i}']$. It follows that for some v_{2i} and v_{2i}', we have $v_{2i} = \mathcal{P}\llbracket e_2 \rrbracket (\widehat{\gamma}[x := v_{1i}]) \equiv_a \mathcal{P}\llbracket e_2 \rrbracket (\widehat{\gamma}[x := v_{1i}']) = v_{2i}'$ for each $i \in \{1, \ldots, n\}$. Thus, we can derive

$$\frac{\Phi_1 = \Phi_1' \qquad \dfrac{v_{21} \equiv_a v_{21}' \quad \cdots \quad v_{2n} \equiv_a v_{2n}'}{w_2 \equiv_a w_2'}}{w_2^{\Phi_1} \equiv_a (w_2')^{\Phi_1'}}$$

  where

$$w_2^{\Phi_1} = \{\mathcal{P}\llbracket e_2 \rrbracket (\widehat{\gamma}[x := v]) \mid v \in w_1\}^{\Phi_1} = \mathcal{P}\llbracket \{e_2 \mid x \in e_1\} \rrbracket \widehat{\gamma}$$

  and similarly

$$(w_2')^{\Phi_1'} = \{\mathcal{P}\llbracket e_2 \rrbracket (\widehat{\gamma}'[x := v]) \mid v \in w_1\}^{\Phi_1} = \mathcal{P}\llbracket \{e_2 \mid x \in e_1\} \rrbracket \widehat{\gamma}'$$

  So we can conclude that $\mathcal{P}\llbracket \{e_2 \mid x \in e_1\} \rrbracket \widehat{\gamma} \equiv_a \mathcal{P}\llbracket \{e_2 \mid x \in e_1\} \rrbracket \widehat{\gamma}'$.

This exhausts all cases and completes the proof. □

5. Static Provenance Analysis

Dynamic provenance may be expensive to compute and nontrivial to implement in a standard relational database system. Moreover, dynamic analysis cannot tell us anything about a query without looking at (annotated) input data. In a typical large database, most of the data is in secondary storage, so it is worthwhile to be able to avoid data access whenever possible. Moreover, even if we want to perform dynamic provenance tracking, a static approximation of dependency information may be useful for optimization. In this section we consider a *static provenance analysis* which statically approximates the dynamic provenance, but can be calculated quickly without accessing the input.

We formulate the analysis as a type-based analysis[Palsberg, 2001]; annotated types (a-types) $\widehat{\tau}$ and raw types (r-types) ω are defined as follows:

$$\widehat{\tau} \quad ::= \quad \omega^{\Phi} \qquad \omega ::= \mathsf{int} \mid \mathsf{bool} \mid \widehat{\tau} \times \widehat{\tau}' \mid \{\widehat{\tau}\}$$

We write $\widehat{\Gamma}$ for a typing context mapping variables to a-types. We lift the auxiliary a-value operations of erasure ($|\widehat{\tau}|$) and annotation extraction ($\|\widehat{\tau}\|$) to a-types as follows:

$$\begin{array}{rcl@{\qquad}rcl}
|\mathsf{int}| &=& \mathsf{int} & \|\mathsf{int}\| &=& \emptyset \\
|\mathsf{bool}| &=& \mathsf{bool} & \|\mathsf{bool}\| &=& \emptyset \\
|\widehat{\tau}_1 \times \widehat{\tau}_2| &=& |\widehat{\tau}_1| \times |\widehat{\tau}_2| & \|\widehat{\tau}_1 \times \widehat{\tau}_2\| &=& \|\widehat{\tau}_1\| \cup \|\widehat{\tau}_2\| \\
|\{\widehat{\tau}\}| &=& \{|\widehat{\tau}|\} & \|\{\widehat{\tau}\}\| &=& \|\widehat{\tau}\| \\
|\omega^{\Phi}| &=& |\omega| & \|\omega^{\Phi}\| &=& \|\omega\| \cup \Phi
\end{array}$$

Moreover, we define compatibility for a-types analogously to compatibility for values, that is, $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are compatible ($\widehat{\tau}_1 \cong \widehat{\tau}_2$) provided $|\widehat{\tau}_1| = |\widehat{\tau}_2|$. Also, we say that an a-type *enriches* an ordinary type τ (written $\widehat{\tau} \gtrsim \tau$) provided $|\widehat{\tau}| = \tau$. These concepts are lifted to a-contexts $\widehat{\Gamma}$ mapping variables to types in the obvious (pointwise) way.

We also define a merge operation $\sqcup$ on compatible types as follows:

$$\begin{array}{rcl}
\mathsf{int} \sqcup \mathsf{int} &=& \mathsf{int} \\
\mathsf{bool} \sqcup \mathsf{bool} &=& \mathsf{bool} \\
(\widehat{\tau}_1 \times \widehat{\tau}_2) \sqcup (\widehat{\tau}_1' \times \widehat{\tau}_2') &=& (\widehat{\tau}_1 \sqcup \widehat{\tau}_1') \times (\widehat{\tau}_2 \sqcup \widehat{\tau}_2') \\
\{\widehat{\tau}\} \sqcup \{\widehat{\tau}'\} &=& \{\widehat{\tau} \sqcup \widehat{\tau}'\} \\
\omega_1^{\Phi_1} \sqcup \omega_2^{\Phi_2} &=& (\omega_1 \sqcup \omega_2)^{\Phi_1 \cup \Phi_2}
\end{array}$$

Finally, we write $\widehat{\tau} \sqsubseteq \widehat{\tau}'$ if $\widehat{\tau}' = \widehat{\tau} \sqcup \widehat{\tau}'$; this is a partial order on types and can be viewed as a subtyping relation.

We interpret a-types $\widehat{\tau}$ as sets of a-values $\widehat{\mathcal{A}}[\![\widehat{\tau}]\!]$. We interpret the annotations in a-types as upper bounds on the annotations in the corresponding a-values:

$$\begin{array}{rcl}
\widehat{\mathcal{A}}[\![\mathsf{int}]\!] &=& \{i \mid i \in \mathbb{Z}\} \\
\widehat{\mathcal{A}}[\![\mathsf{bool}]\!] &=& \{b \mid b \in \mathbb{B}\} \\
\widehat{\mathcal{A}}[\![\widehat{\tau}_1 \times \widehat{\tau}_2]\!] &=& \widehat{\mathcal{A}}[\![\widehat{\tau}_1]\!] \times \widehat{\mathcal{A}}[\![\widehat{\tau}_2]\!] \\
\widehat{\mathcal{A}}[\![\{\widehat{\tau}\}]\!] &=& \mathcal{M}_{\mathsf{fin}}(\widehat{\mathcal{A}}[\![\widehat{\tau}]\!]) \\
\widehat{\mathcal{A}}[\![\omega^{\Phi}]\!] &=& \{w^{\Psi} \mid \Psi \subseteq \Phi, w \in \widehat{\mathcal{A}}[\![\omega]\!]\}
\end{array}$$

The syntactic operations $|-|$, $\|-\|$, $\sqsubseteq$ and $\sqcup$ on types correspond to appropriate semantic operations on sets of a-values. We note some useful properties of these operations:

Lemma 5.1.

1 If $v \in \widehat{\mathcal{A}}[\![\widehat{\tau}]\!]$ then $v \in \mathcal{A}[\![|\widehat{\tau}|]\!]$ and $|v| \in \mathcal{T}[\![|\widehat{\tau}|]\!]$ and $\|v\| \subseteq \|\widehat{\tau}\|$.

$$\frac{x{:}\widehat{\tau} \in \widehat{\Gamma}}{\widehat{\Gamma} \vdash x : \widehat{\tau}} \quad \frac{\widehat{\Gamma} \vdash e_1 : \widehat{\tau}_1 \quad \widehat{\Gamma}, x : \widehat{\tau}_1 \vdash e_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \widehat{\tau}_2}$$

$$\frac{}{\widehat{\Gamma} \vdash i : \mathsf{int}\ \&\ \emptyset} \quad \frac{\widehat{\Gamma} \vdash e_1 : \mathsf{int}\ \&\ \Phi_1 \quad \widehat{\Gamma} \vdash e_2 : \mathsf{int}\ \&\ \Phi_2}{\widehat{\Gamma} \vdash e_1 + e_2 : \mathsf{int}\ \&\ \Phi_1 \cup \Phi_2} \quad \frac{\widehat{\Gamma} \vdash e : \{\mathsf{int}^{\Phi_0}\}\ \&\ \Phi}{\widehat{\Gamma} \vdash \mathsf{sum}(e) : \mathsf{int}\ \&\ \Phi_0 \cup \Phi}$$

$$\frac{}{\widehat{\Gamma} \vdash b : \mathsf{bool}\ \&\ \emptyset} \quad \frac{\widehat{\Gamma} \vdash e : \mathsf{bool}\ \&\ \Phi}{\widehat{\Gamma} \vdash \neg e : \mathsf{bool}\ \&\ \Phi} \quad \frac{\widehat{\Gamma} \vdash e_1 : \mathsf{bool}\ \&\ \Phi_1 \quad \widehat{\Gamma} \vdash e_2 : \mathsf{bool}\ \&\ \Phi_2}{\widehat{\Gamma} \vdash e_1 \wedge e_2 : \mathsf{bool}\ \&\ \Phi_1 \cup \Phi_2}$$

$$\frac{\widehat{\Gamma} \vdash e_1 : \widehat{\tau}_1 \quad \widehat{\Gamma} \vdash e_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash (e_1, e_2) : (\widehat{\tau}_1 \times \widehat{\tau}_2)\ \&\ \emptyset} \quad \frac{\widehat{\Gamma} \vdash e : \omega_1^{\Phi_1} \times \omega_2^{\Phi_2}\ \&\ \Phi}{\widehat{\Gamma} \vdash \pi_i(e) : \omega_i\ \&\ \Phi_i \cup \Phi}\ (i \in \{1,2\})$$

$$\frac{\widehat{\Gamma} \vdash e_1 : \widehat{\tau}_1 \quad \widehat{\Gamma} \vdash e_2 : \widehat{\tau}_2 \quad \widehat{\tau}_1 \cong \widehat{\tau}_2}{\widehat{\Gamma} \vdash e_1 \approx e_2 : \mathsf{bool}\ \&\ \|\widehat{\tau}_1\| \cup \|\widehat{\tau}_2\|} \quad \frac{\widehat{\Gamma} \vdash e_0 : \mathsf{bool}\ \&\ \Phi_0 \quad \widehat{\Gamma} \vdash e_1 : \widehat{\tau}_1 \quad \widehat{\Gamma} \vdash e_2 : \widehat{\tau}_2 \quad \widehat{\tau}_1 \cong \widehat{\tau}_2}{\widehat{\Gamma} \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : (\widehat{\tau}_1 \sqcup \widehat{\tau}_2)^{+\Phi_0}}$$

$$\frac{}{\widehat{\Gamma} \vdash \emptyset : \{\widehat{\tau}\}\ \&\ \emptyset} \quad \frac{\widehat{\Gamma} \vdash e : \widehat{\tau}}{\widehat{\Gamma} \vdash \{e\} : \{\widehat{\tau}\}\ \&\ \emptyset} \quad \frac{\widehat{\Gamma} \vdash e_1 : \{\widehat{\tau}_1\}\ \&\ \Phi_1 \quad \widehat{\Gamma} \vdash e_2 : \{\widehat{\tau}_2\}\ \&\ \Phi_2 \quad \widehat{\tau}_1 \cong \widehat{\tau}_2}{\widehat{\Gamma} \vdash e_1 \cup e_2 : \{\widehat{\tau}_1 \sqcup \widehat{\tau}_2\}\ \&\ \Phi_1 \cup \Phi_2}$$

$$\frac{\widehat{\Gamma} \vdash e_1 : \{\widehat{\tau}_1\}\ \&\ \Phi_1 \quad \widehat{\Gamma}, x{:}\widehat{\tau}_1 \vdash e_2 : \omega\ \&\ \Phi_2}{\widehat{\Gamma} \vdash \{e_2 \mid x \in e_1\} : \{\omega^{\Phi_2}\}\ \&\ \Phi_1} \quad \frac{\widehat{\Gamma} \vdash e : \{\{\widehat{\tau}\}^{\Phi_2}\}\ \&\ \Phi_1}{\widehat{\Gamma} \vdash \bigcup e : \{\widehat{\tau}\}\ \&\ \Phi_1 \cup \Phi_2}$$

$$\frac{\widehat{\Gamma} \vdash e_1 : \{\widehat{\tau}_1\}\ \&\ \Phi_1 \quad \widehat{\Gamma} \vdash e_2 : \{\widehat{\tau}_2\}\ \&\ \Phi_2 \quad \widehat{\tau}_1 \cong \widehat{\tau}_2}{\widehat{\Gamma} \vdash e_1 - e_2 : \{\widehat{\tau}_1\}\ \&\ \|\{\widehat{\tau}_1\}^{\Phi_1}\| \cup \|\{\widehat{\tau}_2\}^{\Phi_2}\|}$$

Fig. 8. Type-based static provenance analysis

2 If $\widehat{\tau}_1 \cong \widehat{\tau}_2$ then $\widehat{\tau}_1 \sqcup \widehat{\tau}_2$ is defined and $\widehat{\mathcal{A}}[\![\widehat{\tau}_1 \sqcup \widehat{\tau}_2]\!] \supseteq \widehat{\mathcal{A}}[\![\widehat{\tau}_1]\!] \cup \widehat{\mathcal{A}}[\![\widehat{\tau}_2]\!]$ and $\|\widehat{\tau}_1 \sqcup \widehat{\tau}_2\| = \|\widehat{\tau}_1\| \cup \|\widehat{\tau}_2\|$.
3 If $\widehat{\tau}_1 \sqsubseteq \widehat{\tau}_2$ then $\widehat{\mathcal{A}}[\![\widehat{\tau}_1]\!] \subseteq \widehat{\mathcal{A}}[\![\widehat{\tau}_2]\!]$ and $\|\widehat{\tau}_1\| \subseteq \|\widehat{\tau}_2\|$.

Figure 8 shows the annotated typing judgment $\widehat{\Gamma} \vdash e : \widehat{\tau}$ (sometimes written $\widehat{\Gamma} \vdash e : \omega\ \&\ \Phi$ for readability, provided $\widehat{\tau} = \omega^{\Phi}$), which extends the plain typing judgment shown in Figure 2.

Proposition 5.1. The judgment $\Gamma \vdash e : \tau$ is derivable if and only if for any $\widehat{\Gamma} \gtrsim \Gamma$, there exists a $\widehat{\tau} \gtrsim \tau$ such that $\widehat{\Gamma} \vdash e : \widehat{\tau}$. Moreover, given $\Gamma \vdash e : \tau$ and $\widehat{\Gamma} \gtrsim \Gamma$, we can compute $\widehat{\tau}$ in polynomial time (by a simple syntax-directed algorithm).

Example 5.1. Consider an annotated type context $\widehat{\Gamma}$, shown in Figure 9(a), where we have annotated field values A, B, C, D, E with colors a, b, c, d, e respectively. Figure 9(b) shows the results of static analysis for the queries in Figure 7. In some cases, the type information simply reflects the field names which are present in the output. However, the colors are not affected by renamings, as in $\rho_{A/C,B/D}$. Furthermore, note that (if we replace the colors a, b, c, d, e with color sets $\{a_1, a_2, a_3\}$, etc.) in each case the type-level colors safely over-approximate the value-level colors calculated in Figure 7.

Example 5.2. To further illustrate the analysis, we consider an extended example for a query that performs grouping and aggregation (equivalent to the one in Example 4.2):

$$Q(R) = \{(\pi_1(x), \mathsf{sum}(G(x))) \mid x \in R\}$$

(a)

$$\widehat{\Gamma} = [R : \{(A : \mathsf{int}^a, B : \mathsf{int}^b)\}, S : \{(C : \mathsf{int}^c, D : \mathsf{int}^d, E : \mathsf{int}^e)\}]$$

(b)

$$\begin{array}{lcl}
\widehat{\Gamma} \vdash \Pi_A(R) & : & \{(A : \mathsf{int}^a)\} \\
\widehat{\Gamma} \vdash \sigma_{A=B}(R) & : & \{(A : \mathsf{int}^a, B : \mathsf{int}^b)\}^{a,b} \\
\widehat{\Gamma} \vdash R \times S & : & \{(A : \mathsf{int}^a, B : \mathsf{int}^b, C : \mathsf{int}^c, D : \mathsf{int}^d, E : \mathsf{int}^e)\} \\
\widehat{\Gamma} \vdash \Pi_{BE}(\sigma_{A=D}(R \times S)) & : & \{(B : \mathsf{int}^b, E : \mathsf{int}^e)\}^{a,d} \\
\widehat{\Gamma} \vdash R \cup \rho_{A/C,B/D}(\Pi_{CD}(S)) & : & \{(A : \mathsf{int}^{a,c}, B : \mathsf{int}^{b,d})\} \\
\widehat{\Gamma} \vdash R - \rho_{A/D,B/E}(\Pi_{DE}(S)) & : & \{(A : \mathsf{int}^a, B : \mathsf{int}^b)\}^{a,b,d,e} \\
\widehat{\Gamma} \vdash \mathsf{sum}(\Pi_A(R)) & : & \mathsf{int}^a \\
\widehat{\Gamma} \vdash \mathsf{count}(R) & : & \mathsf{int} \\
\widehat{\Gamma} \vdash \mathsf{count}(\sigma_{A=B}(R)) & : & \mathsf{int}^{a,b}
\end{array}$$

Fig. 9. (a) Annotated input context (b) Examples of provenance analysis

where we employ the following abbreviations:

$$\begin{array}{rcl}
G(x) & := & \bigcup\{\mathsf{if}\ \pi_1(y) \approx \pi_1(x)\ \mathsf{then}\ \{\pi_2(y)\}\ \mathsf{else}\ \emptyset \mid y \in R\} \\
\widehat{\tau}_R & := & \mathsf{int}^a \times \mathsf{int}^b \\
\widehat{\Gamma} & := & R{:}\{\widehat{\tau}_R\} \\
\widehat{\Gamma}_1 & := & \widehat{\Gamma}, x{:}\widehat{\tau}_R \\
\widehat{\Gamma}_2 & := & \widehat{\Gamma}_1, y{:}\widehat{\tau}_R
\end{array}$$

We will derive $\widehat{\Gamma} \vdash Q(R) : \{\mathsf{int}^a \times \mathsf{int}^{a,b}\}$. The derivation illustrates how color a is propagated to both parts of the result type, while color b is only propagated to the second column.

First, we can reduce the analysis of Q to analyzing $G(x)$ as follows:

$$\dfrac{\widehat{\Gamma} \vdash R : \{\widehat{\tau}_R\} \qquad \dfrac{\dfrac{\widehat{\Gamma}_1 \vdash x : \widehat{\tau}_R}{\widehat{\Gamma}_1 \vdash \pi_1(x) : \mathsf{int}^a} \quad \dfrac{\widehat{\Gamma}_1 \vdash G(x) : \{\mathsf{int}^b\}^a}{\widehat{\Gamma}_1 \vdash \mathsf{sum}(G(x)) : \mathsf{int}^{a,b}}}{\widehat{\Gamma}_1 \vdash (\pi_1(x), \mathsf{sum}(G(x))) : \mathsf{int}^a \times \mathsf{int}^{a,b}}}{\widehat{\Gamma} \vdash \{(\pi_1(x), \mathsf{sum}(G(x))) \mid x \in R\} : \{\mathsf{int}^a \times \mathsf{int}^{a,b}\}}$$

We next reduce the analysis of $G(x)$ to an analysis of the conditional inside $G(x)$:

$$\dfrac{\dfrac{\widehat{\Gamma}_1 \vdash R : \{\widehat{\tau}_R\} \quad \widehat{\Gamma}_2 \vdash \mathsf{if}\ \pi_1(y) \approx \pi_1(x)\ \mathsf{then}\ \{\pi_2(y)\}\ \mathsf{else}\ \emptyset : \{\mathsf{int}^b\}^a}{\widehat{\Gamma}_1 \vdash \{\mathsf{if}\ \pi_1(y) \approx \pi_1(x)\ \mathsf{then}\ \{\pi_2(y)\}\ \mathsf{else}\ \emptyset \mid y \in R\} : \{\{\mathsf{int}^b\}^a\}}}{\widehat{\Gamma}_1 \vdash \bigcup\{\mathsf{if}\ \pi_1(y) \approx \pi_1(x)\ \mathsf{then}\ \{\pi_2(y)\}\ \mathsf{else}\ \emptyset \mid y \in R\} : \{\mathsf{int}^b\}^a}$$

Finally, we can analyze the conditional as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\widehat{\Gamma}_2 \vdash y : \widehat{\tau}_R}}{\widehat{\Gamma}_2 \vdash \pi_1(y) : \mathsf{int}^a} \quad \dfrac{\dfrac{}{\widehat{\Gamma}_2 \vdash x : \widehat{\tau}_R}}{\widehat{\Gamma}_2 \vdash \pi_1(x) : \mathsf{int}^a}}{\widehat{\Gamma}_2 \vdash \pi_1(y) \approx \pi_1(x) : \mathsf{bool}^a} \quad \dfrac{\dfrac{\dfrac{}{\widehat{\Gamma}_2 \vdash y : \widehat{\tau}_R}}{\widehat{\Gamma}_2 \vdash \pi_2(y) : \mathsf{int}^b}}{\widehat{\Gamma}_2 \vdash \{\pi_2(y)\} : \{\mathsf{int}^b\}} \quad \dfrac{}{\widehat{\Gamma}_2 \vdash \emptyset : \{\mathsf{int}\}}}{\widehat{\Gamma}_2 \vdash \mathsf{if}\ \pi_1(y) \approx \pi_1(x)\ \mathsf{then}\ \{\pi_2(y)\}\ \mathsf{else}\ \emptyset : \{\mathsf{int}^b\}^a}$$

5.1. *Correctness of static analysis*

The correctness of the analysis is proved with respect to the provenance-tracking semantics given in Section 4, which we have already shown dependency-correct. Correctness is formulated as a type-soundness theorem, using the refined interpretation $\widehat{\mathcal{A}}[\![-]\!]$ of a-types. Specifically, we show that if $\widehat{\Gamma} \vdash e : \widehat{\tau}$ then $\mathcal{P}[\![e]\!] : \widehat{\mathcal{A}}[\![\widehat{\Gamma}]\!] \to \widehat{\mathcal{A}}[\![\widehat{\tau}]\!]$. Theorem 5.2 immediately implies that the annotations we obtain (statically) by provenance analysis conservatively over-approximate the dependency-correct annotations we obtain (dynamically) by provenance tracking provided the initial value $\widehat{\gamma}$ matches $\widehat{\mathcal{A}}[\![\widehat{\Gamma}]\!]$.

We first establish that the static analysis is a conservative extension of the ordinary type system:

Lemma 5.2. If $\Gamma \vdash e : \tau$ then for any $\widehat{\Gamma} \gtrsim \Gamma$ there exists a $\widehat{\tau} \gtrsim \tau$ such that $\widehat{\Gamma} \vdash e : \widehat{\tau}$.

Proof. Structural induction on derivations; again the only interesting steps are those involving compatibility side-conditions; typically we only need to observe that if $\widehat{\tau}_1, \widehat{\tau}_2 \gtrsim \tau$ then $\widehat{\tau}_1 \cong \widehat{\tau}_2$, so $\widehat{\tau}_1 \sqcup \widehat{\tau}_2$ exists and $\widehat{\tau}_1 \cong \widehat{\tau}_2 \cong \widehat{\tau}_1 \sqcup \widehat{\tau}_2$. □

Lemma 5.3. If $\widehat{\Gamma} \vdash e : \widehat{\tau}$ then $|\widehat{\Gamma}| \vdash e : |\widehat{\tau}|$.

Proof. Straightforward induction on derivations; cases with compatibility side-conditions require observing that by definition $\widehat{\tau}_1 \cong \widehat{\tau}_2 \iff |\widehat{\tau}_1| = |\widehat{\tau}_2|$. □

Lemma 5.4. Every context Γ has at least one enrichment $\widehat{\Gamma} \gtrsim \Gamma$.

Proof. Observe that any type can be lifted to an a-type by annotating each part of it with $\emptyset$. An unannotated context Γ can be lifted to a default annotated context $\widehat{\Gamma}$ by lifting each type. □

Theorem 5.1. The judgment $\Gamma \vdash e : \tau$ is derivable if and only if for any $\widehat{\Gamma}$ enriching Γ, there exists a $\widehat{\tau}$ enriching τ such that $\widehat{\Gamma} \vdash e : \widehat{\tau}$ is derivable for some $\widehat{\tau}$ enriching τ.

Proof. For the forward direction, we use Lemma 5.2. For the reverse direction, suppose the second part holds for a given Γ, e, τ. By Lemma 5.4, we have $\widehat{\Gamma} \vdash e : \widehat{\tau}$ for some $\widehat{\Gamma}$ enriching Γ and $\widehat{\tau}$ enriching τ. Hence by Lemma 5.3, we have $|\widehat{\Gamma}| \vdash e : |\widehat{\tau}|$, but clearly $|\widehat{\Gamma}| = \Gamma$ and $|\widehat{\tau}| = \tau$. □

We next establish useful properties of the a-value operations with respect to the semantics of annotated types:

Lemma 5.5. For any $\Phi, \Psi, \Phi_1, \Phi_2, \widehat{\tau}, \widehat{\tau}_1, \widehat{\tau}_2$:

1 $\widehat{+}$: $\widehat{\mathcal{A}}[\![\mathsf{int}^{\Phi}]\!] \times \widehat{\mathcal{A}}[\![\mathsf{int}^{\Psi}]\!] \to \widehat{\mathcal{A}}[\![\mathsf{int}^{\Phi \cup \Psi}]\!]$.

2 $\widehat{\sum}\colon \widehat{\mathcal{A}}\llbracket\{\mathsf{int}^{\Phi}\}^{\Psi}\rrbracket \to \widehat{\mathcal{A}}\llbracket\mathsf{int}^{\Phi\cup\Psi}\rrbracket$.
3 $\widehat{\neg} : \widehat{\mathcal{A}}\llbracket\mathsf{bool}^{\Phi}\rrbracket \to \widehat{\mathcal{A}}\llbracket\mathsf{bool}^{\Phi}\rrbracket$.
4 $\widehat{\wedge}\colon \widehat{\mathcal{A}}\llbracket\mathsf{bool}^{\Phi}\rrbracket \times \widehat{\mathcal{A}}\llbracket\mathsf{bool}^{\Psi}\rrbracket \to \widehat{\mathcal{A}}\llbracket\mathsf{bool}^{\Phi\cup\Psi}\rrbracket$.
5 $\widehat{\pi}_i : \widehat{\mathcal{A}}\llbracket(\widehat{\tau}_1 \times \widehat{\tau}_2)^{\Phi}\rrbracket \to \widehat{\mathcal{A}}\llbracket\widehat{\tau}_i^{+\Phi}\rrbracket$ for any $i \in \{1,2\}$
6 $\widehat{\approx}\colon \widehat{\mathcal{A}}\llbracket\widehat{\tau}_1\rrbracket \times \widehat{\mathcal{A}}\llbracket\widehat{\tau}_2\rrbracket \to \widehat{\mathcal{A}}\llbracket\mathsf{bool}^{\|\widehat{\tau}_1\|\cup\|\widehat{\tau}_2\|}\rrbracket$.
7 If $\widehat{\tau}_1 \cong \widehat{\tau}_2$ then $\widehat{\mathsf{cond}} : \widehat{\mathcal{A}}\llbracket\mathsf{bool}^{\Phi}\rrbracket \times \widehat{\mathcal{A}}\llbracket\widehat{\tau}_1\rrbracket \times \widehat{\mathcal{A}}\llbracket\widehat{\tau}_2\rrbracket \to \widehat{\mathcal{A}}\llbracket(\widehat{\tau}_1 \sqcup \widehat{\tau}_2)^{+\Phi}\rrbracket$
8 If $\widehat{\tau}_1 \cong \widehat{\tau}_2$ then $\widehat{\cup} : \widehat{\mathcal{A}}\llbracket\{\widehat{\tau}_1\}^{\Phi_1}\rrbracket \times \widehat{\mathcal{A}}\llbracket\{\widehat{\tau}_2\}^{\Phi_2}\rrbracket \to \widehat{\mathcal{A}}\llbracket\{\widehat{\tau}_1 \sqcup \widehat{\tau}_2\}^{\Phi_1\cup\Phi_2}\rrbracket$.
9 If $\widehat{\tau}_1 \cong \widehat{\tau}_2$ then $\widehat{-} : \widehat{\mathcal{A}}\llbracket\{\widehat{\tau}_1\}^{\Phi_1}\rrbracket \times \widehat{\mathcal{A}}\llbracket\{\widehat{\tau}_2\}^{\Phi_2}\rrbracket \to \widehat{\mathcal{A}}\llbracket\{\widehat{\tau}_1\}^{\Phi_1\cup\|\widehat{\tau}_1\|\cup\Phi_2\cup\|\widehat{\tau}_2\|}\rrbracket$.
10 $\widehat{\bigcup} : \widehat{\mathcal{A}}\llbracket\{\{\widehat{\tau}\}^{\Psi}\}^{\Phi}\rrbracket \to \widehat{\mathcal{A}}\llbracket\{\widehat{\tau}\}^{\Phi\cup\Psi}\rrbracket$.

Proof. All of the properties are immediate from the definitions of the operations. □

Theorem 5.2. If $\widehat{\Gamma} \vdash e : \widehat{\tau}$ then $\mathcal{P}\llbracket e\rrbracket : \widehat{\mathcal{A}}\llbracket\widehat{\Gamma}\rrbracket \to \widehat{\mathcal{A}}\llbracket\widehat{\tau}\rrbracket$.

Proof. The proof is by induction on the structure of expressions (and the associated annotated derivations). As before, many of the cases follow immediately by induction and appeals to Lemma 5.5.

— Case $e = x$:

$$\frac{x{:}\widehat{\tau} \in \widehat{\Gamma}}{\widehat{\Gamma} \vdash x : \widehat{\tau}}$$

Note that $\mathcal{P}\llbracket x\rrbracket\widehat{\gamma} = \widehat{\gamma}(x) \in \widehat{\mathcal{A}}\llbracket\widehat{\tau}\rrbracket$ since $\widehat{\gamma} \in \widehat{\mathcal{A}}\llbracket\widehat{\Gamma}\rrbracket$.

— Case $e = (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$:

$$\frac{\widehat{\Gamma} \vdash e_1 : \widehat{\tau}_1 \quad \widehat{\Gamma}, x{:}\widehat{\tau}_1 \vdash e_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \widehat{\tau}_2}$$

By induction on the first subderivation, we have $\mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma} \in \widehat{\mathcal{A}}\llbracket\widehat{\tau}_1\rrbracket$. Hence $\widehat{\gamma}[x := \mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma}] \in \widehat{\mathcal{A}}\llbracket\widehat{\Gamma}, x{:}\widehat{\tau}_1\rrbracket$, so by induction on the second subderivation, we have $\mathcal{P}\llbracket e\rrbracket\widehat{\gamma} = \mathcal{P}\llbracket e_2\rrbracket\widehat{\gamma}([x := \mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma}]) \in \widehat{\mathcal{A}}\llbracket\widehat{\tau}_2\rrbracket$

— Case $e = (e_1, e_2)$:

$$\frac{\widehat{\Gamma} \vdash e_1 : \widehat{\tau}_1 \quad \widehat{\Gamma} \vdash e_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash (e_1, e_2) : (\widehat{\tau}_1 \times \widehat{\tau}_2)\ \&\ \emptyset}$$

By induction, $\mathcal{P}\llbracket e_i\rrbracket\widehat{\gamma} \in \widehat{\mathcal{A}}\llbracket\widehat{\tau}_i\rrbracket$. Thus $(\mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma}, \mathcal{P}\llbracket e_2\rrbracket\widehat{\gamma})^{\emptyset} \in \widehat{\mathcal{A}}\llbracket(\widehat{\tau}_1 \times \widehat{\tau}_2)^{\emptyset}\rrbracket$

— Case $e = \{e'\}$: Similar to the case for pairing.

— Case $e = \{e_2 \mid x \in e_1\}$:

$$\frac{\widehat{\Gamma} \vdash e_1 : \{\widehat{\tau}_1\}\ \&\ \Phi_1 \quad \widehat{\Gamma}, x{:}\widehat{\tau}_1 \vdash e_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash \{e_2 \mid x \in e_1\} : \{\widehat{\tau}_2\}\ \&\ \Phi_1}$$

Let $w^{\Psi} = \mathcal{P}\llbracket e_1\rrbracket\widehat{\gamma}$; then by induction $w^{\Psi} \in \widehat{\mathcal{A}}\llbracket\{\widehat{\tau}_1\}^{\Phi_1}\rrbracket$ and so $w \in \widehat{\mathcal{A}}\llbracket\{\widehat{\tau}_1\}\rrbracket$ and $\Psi \subseteq \Phi_1$. Hence for each $v \in w$, we have $v \in \widehat{\mathcal{A}}\llbracket\widehat{\tau}_1\rrbracket$, so $\widehat{\gamma}[x := v] \in \widehat{\mathcal{A}}\llbracket\widehat{\Gamma}, x{:}\widehat{\tau}_1\rrbracket$. Thus, for each such v, by induction we have $\mathcal{P}\llbracket e_2\rrbracket(\widehat{\gamma}[x := v]) \in \widehat{\mathcal{A}}\llbracket\widehat{\tau}_2\rrbracket$. Moreover, $\mathcal{P}\llbracket e\rrbracket\widehat{\gamma} = \{\mathcal{P}\llbracket e_2\rrbracket(\widehat{\gamma}[x := v]) \mid v \mathrel{\widehat{\in}} w^{\Psi}\} = \{\mathcal{P}\llbracket e_2\rrbracket(\widehat{\gamma}[x := v]) \mid v \in w\}^{\Psi} \in \widehat{\mathcal{A}}\llbracket\{\widehat{\tau}_2\}^{\Phi_1}\rrbracket$ since $\Psi \subseteq \Phi_1$.

□

6. Discussion

We chose to study provenance via the NRC because it is a clean and system-independent core calculus similar to other functional programming languages for which dependence analysis is well-understood. We believe our results can be specialized to common database implementations and physical operators without much difficulty. We have not yet investigated scaling this approach to large datasets or incorporating it into standard relational databases.

We have, however, implemented a prototype NRC interpreter that performs ordinary type-checking and evaluation as well as provenance tracking and analysis. Our prototype currently displays the input and output tables using HTML and uses embedded JavaScript code to highlight backward slices, that is, the parts of the input on which the selected part of the output may depend, according to the analysis. Similarly, the system displays the type information inferred for the query and uses the results of static analysis to highlight relevant parts of the input types for a selected part of the output type.

In the worst (albeit unusual) case, a part of the output could be reported as depending on every part of the input, as a result of spurious dependencies. For example, this is the case for a query such as $(R_1 - R_1) \cup \cdots \cup (R_n - R_n) \cup e$. Of course, this query is equivalent to e and a good query optimizer will recognize this. However, for non-pathological queries encountered in practice, our analysis appears to be reasonably accurate. Even so, typically the structure of the output depends on a large set of locations, such as all of the fields in several columns in the input used in a selection condition; individual fields in the output usually also depend on a smaller number of places from which their values were computed or copied. Thus, implementing provenance tracking in a large-scale database may require developing more efficient representations for large sets of annotations, especially the common case where a part of the output depends on every value in a particular column.

The model we investigate in this article is similar to that of Buneman et al. [2008b] in many respects. There are two salient differences. The first difference is that Buneman et al. [2008b] propagates annotations comprising single (optional) input locations, whereas our approach propagates annotations consisting of *sets* of input locations. The second difference is that our approach provides a strong semantic guarantee formulated in terms of dependence, whereas in contrast the semantics of where-provenance in Buneman et al. [2008b] is an ad hoc syntactic definition justified by a database-theoretic expressiveness result, not a dependency property.

Of these differences, the second is more significant. Their results characterize the possible where-provenance behavior of queries and updates precisely, but tell us little about what might happen if the input is changed. Moreover, their expressiveness results have not yet been extended to handle features such as primitive operations on data values and aggregation. To illustrate the distinction, observe that in the example in Figure 1, the Name fields are copied from the input to the output (thus, they have where-provenance in Buneman et al.'s model) but the AvgMW fields are computed from several sources, not copied (thus, they would have no where-provenance, even though they depend on many parts of the input). We believe the approaches are complementary: each does something useful that the other does not, and in general users may want both kinds of provenance information to be available.

Buneman et al. [2008b] discuss implementing provenance tracking as a source-to-source translation from NRC queries to NRC extended with a new base type color. The idea is to translate ordinary types to types in which each subexpression is paired with an annotation of type color, and translate provenance-tracking queries to ordinary NRC queries over the annotated types that explicitly manage the annotations. This implementation approach has the potential advantage that we can re-use existing query optimization techniques for NRC. A similar query-translation approach should be possible for dependency provenance, by explicitly annotating each part of each value with a set of annotations $\{\mathsf{color}\}$ and using NRC set operations to propagate colors.

Most database systems only implement SQL, which is less expressive than NRC since it does not provide the ability to nest sets as the field values of relations. Nevertheless, it still may be possible to support some annotation-propagation operations within ordinary SQL databases. Suppose we are interested in a particular application in which annotations are numerical timestamps or quality rankings that can be aggregated (e.g. by taking the minimum or maximum). In this case, we can propagate the annotations from the source data to the results according to the provenance semantics. Simple SQL queries can easily be translated to equivalent SQL queries that automatically aggregate annotations in this way, using techniques similar to those used in the DBNotes system [Bhagwat et al., 2005].

For example, consider the query

```
SELECT A, SUM(B) FROM R GROUP BY A
```

over relations $R : \{(A : \mathsf{int}, B : \mathsf{int})\}$. Suppose we have relations $R : \{(A : \mathsf{int}, A_q : \mathsf{int}, B : \mathsf{int}, B_q : \mathsf{int})\}$, in which each field A, B has an accompanying quality rating A_q, B_q. Then we can translate the above SQL query to

```
SELECT A, MIN(A_q), SUM(C), MIN(C_q) FROM R GROUP BY A
```

to associate each value in the output with the minimum quality ranking of the contributing fields—thus, data in the result with a high quality ranking must depend only on high-quality data. However, performing this translation for general SQL queries appears nontrivial. It is well known that flat NRC queries whose input and output types do not involve nested set types and that do not involve grouping or aggregation and map sets of records to sets of records can be translated back to SQL via a normalization process [Wong, 1996], but it is apparently not well-understood how to extract SQL queries from arbitrary NRC expressions involving grouping and aggregation.

We can easily implement static provenance tracking for ordinary SQL queries by translating them to NRC; this does not require changing the database system in any way, since we do not need to execute the queries. Static provenance analysis is slightly more expensive than ordinary typechecking, but since the overhead is proportional only to the size of the schema and query, not the (usually much larger) data, this overhead is minor. Moreover, static analysis may be useful in optimizing provenance tracking, for example by using the results of static analysis to avoid tracking annotations that are statically irrelevant to the output.

Consider for example the following scenario: After running a query, the user identifies an error in the results, and requests a data slice showing the input parts relevant to the error. We can first provide the results of static provenance analysis and show the user which parts of the input database contain data that may have contributed to the error. In a typical relational database, this would narrow things down to the level of database tables and columns, which may be enough for the user to fix the problem. In case this is not specific enough, however, we can still employ the static analysis to speed computing the dynamic provenance. Using the static provenance information, we know that only locations in the input data that correspond to the static provenance of the output location of interest can contribute to that output location. Hence, if we are only interested in the dynamic provenance of a single output location, we might avoid the overhead of dynamically tagging and tracking provenance for parts of the input that we know cannot contribute to the output part of interest. We plan to investigate this potential optimization technique in future work.

6.1. *Comparison with slicing, information flow and other dependence analyses*

The techniques in Section 4 and Section 5 draw upon standard techniques in static analysis [Nielson et al., 2005, Palsberg, 2001]. In particular, the idea of instrumenting the semantics of programs with labels that capture interesting dynamic properties is a well-known technique used in control-flow analysis and information flow control. Moreover, it appears possible to cast our results in the *abstract interpretation* framework [Cousot and Cousot, 1977] that is widely used in static analysis (see e.g. Nielson et al. [2005, ch. 4] for an introduction). Doing so would require adapting abstract interpretation to handle collection types. This does not appear difficult but we preferred to keep the development in this article elementary in order to remain accessible to nonspecialists.

Dependence tracking and analysis have been shown to be useful in many contexts such as program slicing, information-flow security, incremental update of computations, and memoization and caching. A great deal of work has been done on each of these topics, which we cannot completely survey here. We focus on contrasting our work with the most closely related work in these areas.

In *program slicing* [Biswas, 1997, Field and Tip, 1998, Weiser, 1981], the goal is to identify a (small) set of program points whose execution contributes to the value of an output variable (or other observable behavior). This is analogous to our approach to provenance, except that provenance identifies relevant parts of the *input database*, not the *program* (i.e. query). Cheney [2007] discusses the relationship between program slicing and dependency analysis at a high level, complementing the technical details presented in this article.

In computer security, it is often of interest to specify and enforce *information-flow policies* [Sabelfeld and Myers, 2003] that ensure that information marked secret can only be read by privileged users, and that privileged users cannot leak secret information by writing it to public locations. These properties are sometimes referred to as *secrecy* and *integrity*, respectively. Both can be enforced using static (e.g. [Myers, 1999, Volpano et al., 1996] or dynamic (e.g. [Shroff et al., 2008, Jia et al., 2008]) dependency tracking techniques. Our work is closely related to ideas in information flow security, but our goal is not to prevent unauthorized disclosure but instead to explicate the dependencies of the results of a query on its inputs. Nevertheless there are many interesting possible connections that need to be explored, particularly in relating provenance to dynamic information flow tracking [Shroff et al., 2008] and integrating provenance security policies with other access-control, information-flow and audit policies [Swamy et al., 2008, Jia et al., 2008].

In contrast to most work on static analysis and information flow security, we envision the instrumented semantics actually being used to provide feedback to users, rather than only as the basis for proving correctness of a static analysis or preventing security vulnerabilities. This makes our approach closest to (dynamic) slicing. The novelty of our approach with respect to slicing is that we handle a purely functional, terminating database query language and focus on calculating dependencies and slicing information having to do with the (typically large) input data, not the (typically small) query. On the one hand the absence of side-effects, higher-order functions, nontermination, or high-level programming constructs such as objects and modules simplifies some technical matters considerably, and enables more precise information to be tracked; on the other, the presence of collection types and query language constructs leads to new complications not handled in prior work on information flow, slicing or static analysis.

In *self-adjusting computation* [Acar et al., 2008, Acar, 2009], support for efficient incremental recomputation is provided at a language level. Programs are executed using an instrumented semantics that records their dynamic dependencies in a *trace*. Although the first run of the program can be more expensive, subsequent changes to the input can be propagated much more efficiently using the trace. We are currently investigating further applications of ideas from self-adjusting computation to provenance, particularly the use of traces as explanations.

Dependency tracking is also important in *memoization and caching* techniques [Abadi et al., 1996, Acar et al., 2003]. For example, Abadi et al. [1996] study an approach to caching the results of function calls in a software configuration management system, based on a label-propagating operational semantics. Acar et al. [2003] develop a language-based approach to memoizing and caching the results of functional programs. Our work differs from this work in that we contemplate retaining dependency information as an aid to the end-user of a (database) system, not just as an internal data structure used for improving performance.

Our approach to provenance tracking based on dependency analysis has been used in the Fable system [Swamy et al., 2008]. In this work provenance is one of a large class of security policies that can be implemented using Fable, a dependently-typed language for specifying security policies. Subsequently, Swamy et al. [2009] have explored a theory of typed coercions that can be used to implement dependency provenance.

Abadi et al. [1999] argue that techniques such as slicing, information-flow security, and other program analyses such as binding-time analysis can be given a uniform treatment by translating to a common *Dependency Core Calculus*. We believe provenance may also fit into this picture, but in this article, we considered both dynamic and static labeling, whereas the Dependency Core Calculus only allows for static labels. Another difference is that the Dependency Core Calculus is a higher-order, typed lambda-calculus whereas here we have considered the first-order nested relational calculus. It would be interesting to develop a common calculus that can handle both static and dynamic dependence and both higher-order functions and collections, particularly if dynamic information flow and dynamic slicing could also be handled uniformly.

7. Conclusions

Provenance information that relates parts of the result of a database query to relevant parts of the input is useful for many purposes, including judging the reliability of information based on the relevant sources and identifying parts of the database that may be responsible for an error in the output of a query. Although a number of techniques based on this intuition have been proposed, some are ad hoc while others have proven difficult to extend beyond simple conjunctive queries to handle important features of real query languages such as grouping, aggregation, negation and built-in operations.

We have argued that the notion of *dependence*, familiar from program slicing, information flow security, and other program analyses, provides a solid semantic foundation for understanding provenance for complex database queries. In this article we introduced a semantic characterization of *dependency provenance*, showed that minimal dependency provenance is not computable, and presented approximate tracking and analysis techniques. We have also discussed applications of dependency provenance such as computing forward and backward data slices that highlight dependencies between selected parts of the input or output. We have implemented a small-scale prototype to gain a sense of the usefulness and precision of the technique.

We believe there are many promising directions for future work, including implementing efficient practical techniques for large-scale database systems, identifying more sophisticated and useful dependency properties, and studying dependency provenance in other settings such as update languages and workflows.

Acknowledgments

We wish to thank Peter Buneman, Shirley Cohen and Stijn Vansummeren for helpful discussions on this work.

References

Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *ICFP*, pages 83–91, New York, NY, USA, 1996. ACM Press.

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL*, pages 147–160. ACM Press, 1999.

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0.

Umut A. Acar. Self-adjusting computation: (an overview). In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 1–6, 2009.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, pages 14–25, 2003.

Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 309–322, 2008.

Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, pages 953–964, 2006.

Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. *VLDB Journal*, 14(4):373–396, 2005.

S. Biswas. *Dynamic Slicing in Higher-Order Programming Languages*. PhD thesis, University of Pennsylvania, 1997.

Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comp. Sci.*, 149(1):3–48, 1995.

Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *ICDT 2001*, number 1973 in LNCS, pages 316–330. Springer, 2001.

Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. On propagation of deletions and annotations through views. In *PODS*, pages 150–158, 2002.

Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *SIGMOD 2006*, pages 539–550, 2006.

Peter Buneman, James Cheney, Wang-Chiew Tan, and Stijn Vansummeren. Curated databases. In *Proceedings of the 2008 Symposium on Principles of Database Systems (PODS 2008)*, pages 1–12, 2008a. Invited paper.

Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems*, 33(4):28, November 2008b.

James Cheney. Program slicing and data provenance. *IEEE Data Engineering Bulletin*, pages 22–28, December 2007.

James Cheney, Amal Ahmed, and Umut A. Acar. Provenance as dependency analysis. In M. Arenas and M. I. Schwartzbach, editors, *Proceedings of the 11th International Symposium on Database Programming Languages (DBPL 2007)*, number 4797 in LNCS, pages 139–153. Springer-Verlag, 2007.

James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

John Field and Frank Tip. Dynamic dependence in term rewriting systems and its application to program slicing. *Information and Software Technology*, 40(11–12):609–636, November/December 1998.

Ian Foster and Luc Moreau, editors. *Proceedings of the 2006 International Provenance and Annotation Workshop (IPAW 2006).* Number 4145 in LNCS. Springer-Verlag, 2006.

J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated XML: queries and provenance. In *PODS*, pages 271–280, 2008.

Floris Geerts, Anastasios Kementsietsidis, and Diego Milano. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE 2006*, page 82, 2006.

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40. ACM Press, 2007.

Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. A formal model of dataflow repositories. In Sarah Cohen Boulakia and Val Tannen, editors, *DILS*, volume 4544 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2007.

Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: a programming language for authorization and audit. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 27–38, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. .

C. Lynch. Authenticity and integrity in the digital environment: An exploratory analysis of the central role of trust. In *Authenticity in the Digital Environment*. CLIR, 2000. CLIR Report pub92; ISBN 1-887334-77-7.

Luc Moreau, Bertram Ludäscher, Ilkay Altintas, Roger S. Barga, Shawn Bowers, Steven Callahan, George Chin Jr., Ben Clifford, Shirley Cohen, Sarah Cohen-Boulakia, Susan Davidson, Ewa Deelman, Luciano Digiampietri, Ian Foster, Juliana Freire, James Frew, Joe Futrelle, Tara Gibson, Yolanda Gil, Carole Goble, Jennifer Golbeck, Paul Groth, David A. Holland, Sheng Jiang, Jihie Kim, David Koop, Ales Krenek, Timothy McPhillips, Gaurang Mehta, Simon Miles, Dominic Metzger, Steve Munroe, Jim Myers, Beth Plale, Norbert Podhorszki, Varun Ratnakar, Emanuele Santos, Carlos Scheidegger, Karen Schuchardt, Margo Seltzer, Yogesh L. Simmhan, Claudio Silva, Peter Slaughter, Eric Stephan, Robert Stevens, Daniele Turi, Huy Vo, Mike Wilde, Jun Zhao, and Yong Zhao. The First Provenance Challenge. *Concurrency and Computation: Practice and Experience*, 20(5):409–418, 2007.

Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *USENIX*, pages 43–56, Boston, MA, June 2006. USENIX.

Andrew C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3. .

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, second edition, 2005.

Jens Palsberg. Type-based analysis and applications. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 20–27, New York, NY, USA, 2001. ACM. ISBN 1-58113-413-4. .

Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

Paritosh Shroff, Scott F. Smith, and Mark Thober. Securing information flow via dynamic capture of dependencies. *J. Comput. Secur.*, 16(5):637–688, 2008. ISSN 0926-227X.

Yogesh Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.

Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE Symposium on Security and Privacy*, pages 369–383. IEEE Computer Society, 2008.

Nikhil Swamy, Michael W. Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 329–340. ACM, 2009. ISBN 978-1-60558-332-7.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, 1996. ISSN 0926-227X.

P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

Y. Richard Wang and Stuart E. Madnick. A polygen model for heterogeneous database systems: The source tagging perspective. In *VLDB*, pages 519–538, 1990.

Mark Weiser. Program slicing. In *ICSE*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

Limsoon Wong. Normal forms and conservative extension properties for query languages over collection types. *Journal of Computer and System Sciences*, 52(3):495–505, 1996.

A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE 1997*, pages 91–102, 1997.