Provenance Traces

Extended Report

James Cheney

University of Edinburgh
jcheney@inf.ed.ac.uk

Umut A. Acar   Amal Ahmed

Toyota Technological Institute, Chicago
[umut|amal]@tti-c.org

Abstract

Provenance is information about the origin, derivation, ownership, or history of an object. It has recently been studied extensively in scientific databases and other settings due to its importance in helping scientists judge data validity, quality and integrity. However, most models of provenance have been stated as ad hoc definitions motivated by informal concepts such as "comes from", "influences", "produces", or "depends on". These models lack clear formalizations describing in what sense the definitions capture these intuitive concepts. This makes it difficult to compare approaches, evaluate their effectiveness, or argue about their validity.

We introduce *provenance traces*, a general form of provenance for the *nested relational calculus* (NRC), a core database query language. Provenance traces are related to the traces that have been used in self-adjusting computation, but differ in important respects. We define a tracing operational semantics for NRC queries that produces both an ordinary result and a trace of the execution. These traces satisfy two strong semantic guarantees: *consistency*, meaning that the traces describe what actually happened during execution, and *fidelity*, meaning that the traces are faithful to the original expression even if the input is changed. We show that three pre-existing forms of provenance for the NRC can be extracted from provenance traces. Thus, provenance traces provide a general semantic foundation for comparing and unifying models of provenance in databases. We also discuss *slicing* techniques that extract a smaller trace that contains only information pertinent to part of the output.

1. Introduction

Sophisticated computer systems and programming techniques, particularly database management systems and distributed computation, are now being used for large-scale scientific endeavors in many fields including biology, physics and astronomy. They are typically being used directly by scientists, not professional programmers. For these users, the behavior of such systems is opaque and — often correctly — viewed as unreliable. Simply presenting the result of a computation is not considered sufficient to establish its repeatability or scientific value in (for example) a journal article. Instead, it is considered essential to provide high-level explanations of how a part of the result of a database query or distributed computation was derived from its inputs, or how a database came to be the way it is. Such information about the source, context, derivation, or history of a (data) object is often called *provenance*.

Currently, many systems either require their users to deal with provenance manually or provide one of a variety of ad hoc, custom solutions. Manual recordkeeping is tedious and error-prone, while both manual and custom solutions are expensive and provide few formal correctness guarantees. This state of affairs strongly motivates research into automatic and standardized techniques for recording, managing, and exploiting provenance in databases and other systems.

A number of approaches for automatic provenance tracking have been studied, each aiming to capture some intuitive aspect of provenance such as "Where did a result come from in the input?" (Buneman et al. 2001), "What inputs influenced a result?"(Cui et al. 2000; Buneman et al. 2001), "How was a result produced from the input?" (Green et al. 2007), or "What inputs do results depend on?" (Cheney et al. 2007). However, there is not yet much understanding of the advantages, disadvantages and formal guarantees offered by each, or of the relationships among them. Many of these techniques have been presented as ad hoc definitions without clear formal specifications of the problem the definitions are meant to solve. In some cases, loose specifications have been developed, but they appear difficult to extend beyond simple settings such as monotone relational queries.

We believe that semantic foundations are needed for provenance in order to understand and relate existing techniques, as well as to motivate and validate new techniques. We focus on provenance in database management systems, because of its practical importance and because several interesting provenance techniques have already been developed in this setting. We investigate a semantic foundation for provenance in databases based on *traces*. We begin with an operational semantics based on stores in which each part of each value has a label. We instrument the semantics so that as an expression evaluates, we record certain properties of the operational derivation in a *provenance trace*. Provenance traces record the relationships between the labels in the store, ultimately linking the result of a computation to the input. Traces can be viewed as a concrete representation of the operational semantics derivation showing how each part of the output was computed from the input and intermediate values.

We employ the *nested relational calculus* (NRC), a core database query language closely related to monadic comprehensions as used in Haskell and other functional programming languages (Wadler 1992). The nested relational model also forms the basis for distributed programming systems such as MapReduce (Dean and Ghemawat 2008) and PigLatin (Olston et al. 2008) and is closely related to XML. Thus, our results should generalize to these other settings.

This paper makes the following contributions:
- We define traces, traced evaluation for NRC queries, and a trace adaptation semantics and establish that traces provide a solid semantic foundation for provenance. Specifically, we show that the trace generated by evaluating an expression is consistent with the resulting store, and that such traces are faithful to the original expression even if the input store is changed.
- We show that we can extract several other forms of provenance that have been developed for the NRC from traces, including *where-provenance* (Buneman et al. 2001, 2007),

dependency provenance (Cheney et al. 2007), and *semiring-provenance* (Green et al. 2007; Foster et al. 2008). The semiring-provenance model already generalizes several other forms of provenance such as *why-provenance* (Buneman et al. 2001) and *lineage* (Cui et al. 2000), but where-provenance and dependency-provenance are not instances of the semiring model. Provenance traces thus unify three previously unrelated provenance models.

Since traces are usually large and difficult to interpret, we discuss slicing and simplification techniques for extracting smaller traces that are relevant to a part of the output. These slicing techniques are motivated by notions of dependence similar to those used as a foundation for program slicing, information flow security, and access control (Abadi et al. 1999). However, a full investigation of provenance trace slicing is beyond the scope of this paper.

The traces used in this paper are also related to traces studied in other settings, particularly in AFL, an adaptive functional language introduced by Acar et al. (2006). However, there are important differences. First, while AFL leaves it up to the programmer to identify *modifiable* inputs and *changeable* outputs, provenance traces implicitly treat every part of the input as modifiable and every part of the output as changeable. This may make provenance traces too inefficient for practical use, but our main goal here is to identify a rich, principled form of provenance and efficiency is a secondary concern. Second, AFL traces are based directly on source language expressions, and were not designed with human-readability or provenance extraction in mind. In contrast, provenance traces can be viewed as directed acyclic graphs (with some extra structure and annotations) that can easily be traversed to extract other forms of provenance. Finally, AFL includes user-defined, recursive functions, whereas the NRC does not include function definitions but does provide collection types and comprehension operations. These differences are minor; it appears straightforward to add the missing features to the respective languages.

Synopsis The structure of the rest of this paper is as follows. Section 2 reviews the nested relational calculus, and introduces an operational, destination-passing, store-based semantics for NRC. Section 3 defines provenance traces and introduces a traced operational semantics for NRC queries and a trace adaptation semantics for adjusting traces to changes to the input. Section 4 establishes the key metatheoretic and semantic properties of traces. Section 5 discusses extracting other forms of provenance from traces, and Section 6 briefly discusses trace slicing and simplification techniques. We discuss related and future work and conclude in Sections 7–8.

2. Nested relational calculus

The nested relational calculus (Buneman et al. 1995) is a simply-typed core language (closely related to monadic comprehensions (Wadler 1992)) that is as expressive as standard database query languages such as SQL but has simpler syntax and cleaner semantics. (We do not address certain dark corners of SQL such as NULL values.) The syntax of NRC types $\tau \in Type$ is as follows:

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

Types include base types such as int and bool, pairing types $\tau_1 \times \tau_2$, and collection types $\{\tau\}$. Collection types $\{\tau\}$ are often taken to be sets, bags (multisets), or lists; in this paper, we consider multiset collections only. We omit first-class function types and λ-terms because most database systems do not support them.

We assume countably infinite, disjoint sets *Var* of *variables* and *labels Lab*. The syntax of NRC expressions $e \in Exp$ is as follows:

$$
\begin{aligned}
e \quad ::= \quad & l \mid x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid (e_1, e_2) \mid \pi_i(e) \\
& \mid\ b \mid \neg e \mid e_1 \wedge e_2 \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
& \mid\ i \mid e_1 + e_2 \mid \sum\{e_2 \mid x \in e_1\} \mid e_1 \approx e_2 \\
& \mid\ \emptyset \mid \{e\} \mid e_1 \cup e_2 \mid \bigcup\{e_2 \mid x \in e_1\} \mid \mathsf{empty}(e)
\end{aligned}
$$

Variables and let-expressions, pairing, boolean, and integer operations are standard. Labels are used in the operational semantics (Section 2.4). The expression $\emptyset$ denotes the empty collection; $\{e\}$ constructs a singleton collection, $e_1 \cup e_2$ takes the (multiset) union of two collections, and $\bigcup\{e \mid x \in e_0\}$ iterates over a collection obtained by evaluating e, applying $e(x)$ to each element of the collection, and unioning the results. Note that we can define $\{e \mid x \in e_0\}$ as $\bigcup\{\{e\} \mid x \in e_0\}$. Finally, the $\mathsf{empty}(e)$ predicate tests whether the collection denoted by e is empty, and the $\sum\{e \mid x \in e_0\}$ operation takes the sum of a collection of integers.

Expressions are identified modulo alpha-equivalence, regarding x bound in $e(x)$ in the expressions $\bigcup\{e(x) \mid x \in e_0\}$, $\sum\{e(x) \mid x \in e_0\}$ and $\mathsf{let}\ x = e_0\ \mathsf{in}\ e(x)$. We write $e[l/x]$ for the result of substituting a label l for a variable x in e; labels cannot be bound so substitution is naturally capture-avoiding.

2.1 Examples

As with many core languages, it is inconvenient to program directly in NRC. Instead, it is often more convenient to use idiomatic "comprehension syntax" similar to Haskell's list comprehensions (Wadler 1992; Buneman et al. 1994). These can be viewed as syntactic sugar for primitive NRC expressions, just as in Haskell list comprehensions can be translated to the primitive monadic operations on lists. Although we use unlabeled pairs, the NRC can also be extended easily with convenient named-record syntax. These techniques are standard so here we only illustrate them via examples which will be used later in the paper.

Example 1 Suppose we have relations $R : \{(A{:}\mathsf{int}, B{:}\mathsf{int}, C{:}\mathsf{int})\}$, $S : \{(C{:}\mathsf{int}, D{:}\mathsf{int})\}$. Consider the SQL "join" query

```
SELECT R.A,R.B,S.D FROM R,S WHERE R.C = S.C
```

This is equivalent to the core NRC expression

$$
\begin{aligned}
Q_1 \quad = \quad & \bigcup\{\bigcup\{\mathsf{if}\ r.C = s.C \\
& \qquad\qquad \mathsf{then}\ \{(A{:}r.A, B{:}r.B, D{:}s.D)\}\ \mathsf{else}\ \emptyset \\
& \quad \mid s \in S\} \mid r \in R\}
\end{aligned}
$$

Example 2 Given R, S as above, the SQL "aggregation" query

```
SELECT 42 AS C, SUM(D) FROM S WHERE C = 2
UNION
SELECT B AS C, A AS D FROM R WHERE C = 4
```

can be expressed as

$$
\begin{aligned}
Q_2 \quad = \quad & \{(C : 42, D : \sum\{\mathsf{if}\ s.C = 2\ \mathsf{then}\ s.D\ \mathsf{else}\ 0 \mid s \in S\})\} \\
& \cup \quad \bigcup\{\mathsf{if}\ r.C = 4\ \mathsf{then}\ \{(C{:}r.B, D{:}r.A)\}\ \mathsf{else}\ \emptyset \mid r \in R\}
\end{aligned}
$$

Some sample input tables and the results of running Q_1 and Q_2 on them are shown in Figure 1. The labels $r, r_1, \ldots$ in are used in the operational semantics, as discussed in Section 2.4.

2.2 Type system

NRC expressions can be typechecked using standard techniques. The typechecking rules are shown in Figure 2. We employ contexts Γ of the form $\Gamma ::= \cdot \mid \Gamma, x{:}\tau$.

2.3 Denotational semantics

The semantics of NRC expressions is usually defined denotationally. We consider values $v \in Val$ of the form:

$$v ::= i \mid b \mid (v_1, v_2) \mid \{v_1, \ldots, v_n\}$$

Figure 1. Examples

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

$$\frac{i \in \mathbb{Z}}{\Gamma \vdash i : \text{int}} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

$$\frac{b \in \mathbb{B}}{\Gamma \vdash b : \text{bool}} \quad \frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \neg e : \text{bool}} \quad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \wedge e_2 : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \approx e_2 : \text{bool}} \quad \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$$

$$\frac{\Gamma \vdash e : \{\tau\}}{\Gamma \vdash \text{empty}(e) : \text{bool}} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i(e) : \tau_i}$$

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

$$\frac{\Gamma \vdash e_0 : \{\tau_0\} \quad \Gamma, x{:}\tau_0 \vdash e : \{\tau\}}{\Gamma \vdash \bigcup\{e \mid x \in e_0\} : \{\tau\}} \quad \frac{\Gamma \vdash e_0 : \{\tau_0\} \quad \Gamma, x{:}\tau_0 \vdash e : \text{int}}{\Gamma \vdash \sum\{e \mid x \in e_0\} : \text{int}}$$

Figure 2. Expression well-formedness

where $i \in \mathbb{Z}$ and $b \in \mathbb{B}$, and interpret types as sets of values, as follows:

$$\begin{aligned}
\llbracket \text{int} \rrbracket &= \mathbb{Z} = \{\ldots, -1, 0, 1, 2, \ldots\} \\
\llbracket \text{bool} \rrbracket &= \mathbb{B} = \{\mathsf{t}, \mathsf{f}\} \\
\llbracket \tau_1 \times \tau_2 \rrbracket &= \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket \\
\llbracket \{\tau\} \rrbracket &= \mathcal{M}_{\text{fin}}(\llbracket \tau \rrbracket)
\end{aligned}$$

We write $\mathcal{M}_{\text{fin}}(X)$ for the set of *finite multisets* of values. Figure 3 shows the (standard) equations defining the denotational semantics of NRC expressions. NRC does not include arbitrary recursive definitions, so we do not need to deal with nontermination.

We write $\gamma : Var \to Val$ for a finite function (or environment) mapping variables x to values v. We write $\llbracket \Gamma \rrbracket$ for the set of all environments γ such that $\gamma(x) \in \llbracket \Gamma(x) \rrbracket$ for all $x \in \text{dom}(\gamma)$.

The type system given above is sound in the following sense:

Proposition 1. *If* $\Gamma \vdash e : \tau$ *then* $\llbracket e \rrbracket : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket$.

2.4 Operational semantics

The semantics of NRC is usually presented denotationally. For the purposes of this paper, we will introduce an operational semantics

$$\begin{aligned}
\llbracket x \rrbracket \gamma &= \gamma(x) \\
\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket \gamma &= \llbracket e_2 \rrbracket \gamma[x \mapsto \llbracket e_1 \rrbracket \gamma] \\
\llbracket i \rrbracket \gamma &= i \\
\llbracket e_1 + e_2 \rrbracket \gamma &= \llbracket e_1 \rrbracket \gamma + \llbracket e_2 \rrbracket \gamma \\
\llbracket \sum\{e \mid x \in e_0\} \rrbracket \gamma &= \sum\{\llbracket e \rrbracket \gamma[x \mapsto v] \mid v \in \llbracket e_0 \rrbracket \gamma\} \\
\llbracket b \rrbracket \gamma &= b \\
\llbracket \neg e \rrbracket \gamma &= \neg \llbracket e \rrbracket \gamma \\
\llbracket e_1 \wedge e_2 \rrbracket \gamma &= \llbracket e_1 \rrbracket \gamma \wedge \llbracket e_2 \rrbracket \gamma \\
\llbracket (e_1, e_2) \rrbracket \gamma &= (\llbracket e_1 \rrbracket \gamma, \llbracket e_2 \rrbracket \gamma) \\
\llbracket \pi_i(e) \rrbracket \gamma &= \pi_i(\llbracket e \rrbracket \gamma) \\
\llbracket \emptyset \rrbracket \gamma &= \emptyset \\
\llbracket \{e\} \rrbracket \gamma &= \{\llbracket e \rrbracket \gamma\} \\
\llbracket e_1 \cup e_2 \rrbracket \gamma &= \llbracket e_1 \rrbracket \gamma \sqcup \llbracket e_2 \rrbracket \gamma \\
\llbracket \bigcup\{e \mid x \in e_0\} \rrbracket \gamma &= \bigsqcup\{\llbracket e \rrbracket \gamma[x \mapsto v] \mid v \in \llbracket e_0 \rrbracket \gamma\} \\
\llbracket \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \rrbracket \gamma &= \begin{cases} \llbracket e_1 \rrbracket \gamma & \text{if } \llbracket e_0 \rrbracket \gamma = \mathsf{t} \\ \llbracket e_2 \rrbracket \gamma & \text{if } \llbracket e_0 \rrbracket \gamma = \mathsf{f} \end{cases} \\
\llbracket e_1 \approx e_2 \rrbracket \gamma &= \begin{cases} \mathsf{t} & \text{if } \llbracket e_1 \rrbracket \gamma = \llbracket e_2 \rrbracket \gamma \\ \mathsf{f} & \text{if } \llbracket e_1 \rrbracket \gamma \neq \llbracket e_2 \rrbracket \gamma \end{cases} \\
\llbracket \text{empty}(e) \rrbracket \gamma &= \begin{cases} \mathsf{t} & \text{if } \llbracket e \rrbracket \gamma = \emptyset \\ \mathsf{f} & \text{if } \llbracket e \rrbracket \gamma \neq \emptyset \end{cases}
\end{aligned}$$

Figure 3. Denotational semantics of NRC

based on *stores* in which every part of every value has a label. This semantics will serve as the basis for our trace semantics, since labels can easily be used to address parts of the input and output of a query; this is challenging to accommodate in a denotational semantics. We define *value constructors* $k \in Con$ as follows:

$$k ::= i \mid b \mid (l_1, l_2) \mid \{l_1 : m_1, \ldots, l_n : m_n\}$$

Here, $\{l_1 : m_1, \ldots, l_n : m_n\}$ denotes a multiset of labels (often denoted L, L'), where m_i is the multiplicity of l_i. Multiplicities are assumed nonzero and omitted when equal to 1. Multisets are equivalent up to reordering and we assume the elements l_i are distinct. We write $M \sqcup N$ for multiset union and $M \oplus N$ for domain-disjoint multiset union, defined only when $\text{dom}(M) \cap \text{dom}(N) = \emptyset$.

We write $Lab(k)$ for the set of labels mentioned in k. Stores are finite maps $\sigma : Lab \to Con$ from labels to constructors. We also consider label environments to be finite maps from variables to labels $\gamma : Var \to Lab$.

We will restrict attention to NRC expressions in "A-normal form", defined as follows:

$$\begin{aligned}
w \quad &::= \quad x \mid l \\
e \quad &::= \quad w \mid \text{let } x = e_1 \text{ in } e_2 \mid (w_1, w_2) \mid \pi_i(w) \\
&\quad\mid\quad b \mid \neg w \mid w_1 \wedge w_2 \mid \text{if } w_0 \text{ then } e_1 \text{ else } e_2 \\
&\quad\mid\quad i \mid w_1 + w_2 \mid \sum\{e_2 \mid x \in w_1\} \mid w_1 \approx w_2 \\
&\quad\mid\quad \emptyset \mid \{w\} \mid w_1 \cup w_2 \mid \bigcup\{e_2 \mid x \in w_1\} \mid \text{empty}(w)
\end{aligned}$$

The normalization translation is standard and straightforward, so omitted. The operational semantics rules are shown in Figure 5. The rules are in destination-passing style. We use two judgments: $\sigma, l \Leftarrow e \Downarrow \sigma'$, meaning "in store σ, evaluating e at location l yields store σ'"; and $\sigma, x{\in}L, e \Downarrow^\star \sigma', L'$, meaning "in store σ, iterating e with x bound to each element of L yields store σ' and result labels L'." The second judgment deals with iteration over

$$
\begin{aligned}
\mathrm{op}(l, \sigma) &= \sigma(l) \\
\mathrm{op}(i, \sigma) &= i \\
\mathrm{op}(l_1 + l_2, \sigma) &= \sigma(l_1) +_{\mathbb{Z}} \sigma(l_2) \\
\mathrm{op}(l_1 \approx l_2, \sigma) &= \begin{cases} \mathsf{t} & (\sigma(l_1) = \sigma(l_2)) \\ \mathsf{f} & (\sigma(l_1) \neq \sigma(l_2)) \end{cases} \\
\mathrm{op}(b, \sigma) &= b \\
\mathrm{op}(l_1 \wedge l_2, \sigma) &= \sigma(l_1) \wedge_{\mathbb{B}} \sigma(l_2) \\
\mathrm{op}(\neg l, \sigma) &= \neg_{\mathbb{B}} \sigma(l) \\
\mathrm{op}((l_1, l_2), \sigma) &= (l_1, l_2) \\
\mathrm{op}(\emptyset, \sigma) &= \emptyset \\
\mathrm{op}(\{l\}, \sigma) &= \{l : 1\} \\
\mathrm{op}(l_1 \cup l_2, \sigma) &= \sigma(l_1) \sqcup \sigma(l_2) \\
\mathrm{op}(\mathsf{empty}(l), \sigma) &= \begin{cases} \mathsf{t} & (\sigma(l) = \emptyset) \\ \mathsf{f} & (\sigma(l) \neq \emptyset) \end{cases}
\end{aligned}
$$

Figure 4. Definition of op

$$
\frac{}{\sigma, l \Leftarrow t \Downarrow \sigma[l := \mathrm{op}(t, \sigma)]}
$$

$$
\frac{\sigma, l' \Leftarrow e_1 \Downarrow \sigma' \quad \sigma', l \Leftarrow e_2[l'/x] \Downarrow \sigma'' \quad l' \text{ fresh}}{\sigma, l \Leftarrow \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow \sigma''}
$$

$$
\frac{\sigma(l') = b \quad \sigma, l \Leftarrow e_b \Downarrow \sigma'}{\sigma, l \Leftarrow \mathsf{if}\ l'\ \mathsf{then}\ e_{\mathsf{t}}\ \mathsf{else}\ e_{\mathsf{f}} \Downarrow \sigma'} \qquad \frac{\sigma(l') = (l_1, l_2)}{\sigma, l \Leftarrow \pi_i(l') \Downarrow \sigma[l := \sigma(l_i)]}
$$

$$
\frac{\sigma, x{\in}\sigma(l_0), e \Downarrow^{\star} \sigma', L}{\sigma, l \Leftarrow \bigcup\{e \mid x \in l_0\} \Downarrow \sigma'[l := \bigsqcup \sigma'[L']]}
$$

$$
\frac{\sigma, x{\in}\sigma(l_0), e \Downarrow^{\star} \sigma', L}{\sigma, l \Leftarrow \sum\{e \mid x \in l_0\} \Downarrow \sigma'[l := \sum \sigma'[L']]}
$$

$$
\frac{\sigma, l' \Leftarrow e[l/x] \Downarrow \sigma' \quad l' \text{ fresh}}{\sigma, x{\in}\emptyset, e \Downarrow^{\star} \sigma, \emptyset \quad \sigma, x{\in}\{l : m\}, e \Downarrow^{\star} \sigma', \{l' : m\}}
$$

$$
\frac{\sigma, x{\in}L_1, e \Downarrow^{\star} \sigma_1, L_1' \quad \sigma, x{\in}L_2, e \Downarrow^{\star} \sigma_2, L_2'}{\sigma, x{\in}L_1 \oplus L_2, e \Downarrow^{\star} \sigma_1 \uplus_\sigma \sigma_2, L_1' \oplus L_2'}
$$

Figure 5. Operational semantics

multisets involved in comprehensions; this exemplifies a common pattern used throughout the paper.

Many of the rules are similar; for brevity, we use a single rule for *terms t* of the following forms:

$$
\begin{aligned}
t \quad ::= \quad & i \mid l_1 + l_2 \mid l_1 \approx l_2 \mid b \mid \neg l \mid l_1 \wedge l_2 \\
& \mid \quad (l_1, l_2) \mid l \mid \emptyset \mid \{l\} \mid l_1 \cup l_2 \mid \mathsf{empty}(l)
\end{aligned}
$$

Each term is either a constant, a label, or a constructor or primitive function applied to some labels. The meaning of each of these operations is defined via the op function, as shown in Figure 4, which maps a term $t \in \textit{Term}$ and a store $\sigma : \textit{Lab} \to \textit{Con}$ to a constructor.

When L is a set of labels, we write $\sigma[L]$ for the multiset of constructors $\{\sigma(l) : m \mid l : m \in L\}$. This notation is used in the rules for $\bigcup$ and $\sum$. In this notation, the standard definition of summation of multisets of integers is $\sum\{i_1 : m_1, \ldots, i_n : m_n\} = \sum_{j=1}^n i_j \cdot m_j$. Similarly, $\bigsqcup\{L_1 : m_1, \ldots, L_n : m_n\} = m_1 \cdot L_1 \sqcup \cdots \sqcup m_n \cdot L_n$, where $m \cdot \{l_1 : k_1, \ldots, l_n : k_n\} = \{l_1 : m \cdot k_1, \ldots, m \cdot k_n\}$.

The iteration rules $\sigma, x{\in}L, e \Downarrow^{\star} \sigma', L'$, evaluate e with x bound to each $l \in L$ independently, preserving the multiplicity of labels. They split L using $\oplus$ and combine the result stores using the orthogonal store merging operation $\uplus_\sigma$ defined as follows:

Definition 1 (Orthogonal extensions and merging) We say σ_1 and σ_2 are *orthogonal extensions* of σ if $\sigma_1 = \sigma \uplus \sigma_1'$ and $\sigma_2 = \sigma \uplus \sigma_2'$ and $\mathrm{dom}(\sigma_1') \cap \mathrm{dom}(\sigma_2') = \emptyset$, and we write $\sigma_1 \uplus_\sigma \sigma_2$ for $\sigma \uplus \sigma_1' \uplus \sigma_2'$.

$$
\frac{\Omega(w_1) = \Omega(w_2) = \mathsf{int}}{\Omega \vdash_{\mathsf{term}} i : \mathsf{int}} \quad \frac{\Omega(w_1) = \Omega(w_2) = \mathsf{int}}{\Omega \vdash_{\mathsf{term}} w_1 + w_2 : \mathsf{int}} \quad \frac{}{\Omega \vdash_{\mathsf{term}} w_1 \approx w_2 : \mathsf{bool}}
$$

$$
\frac{}{\Omega \vdash_{\mathsf{term}} (w_1, w_2) : \Omega(w_1) \times \Omega(w_2)}
$$

$$
\frac{\Omega(w_1) = \Omega(w_2) = \mathsf{bool}}{\Omega \vdash_{\mathsf{term}} b : \mathsf{bool}} \quad \frac{\Omega(w_1) = \Omega(w_2) = \mathsf{bool}}{\Omega \vdash_{\mathsf{term}} w_1 \wedge w_2 : \mathsf{bool}} \quad \frac{\Omega(w) = \mathsf{bool}}{\Omega \vdash_{\mathsf{term}} \neg w : \mathsf{bool}}
$$

$$
\frac{\Omega(w) = \tau}{\Omega \vdash_{\mathsf{term}} \emptyset : \{\tau\}} \quad \frac{\Omega(w) = \tau}{\Omega \vdash_{\mathsf{term}} \{w\} : \{\tau\}} \quad \frac{\Omega(w_1) = \{\tau\} = \Omega(w_2)}{\Omega \vdash_{\mathsf{term}} w_1 \cup w_2 : \{\tau\}}
$$

$$
\frac{\Omega(w) = \{\tau\}}{\Omega \vdash_{\mathsf{term}} \mathsf{empty}(w) : \mathsf{bool}} \quad \frac{}{\Omega \vdash_{\mathsf{term}} w : \Omega(w)}
$$

$$
\frac{\Omega \vdash_{\mathsf{term}} t : \tau}{\Omega \vdash t : \tau} \quad \frac{\Omega \vdash e_1 : \tau' \quad \Omega, x{:}\tau' \vdash e_2 : \tau}{\Omega \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}
$$

$$
\frac{\Omega(w) = \tau_1 \times \tau_2}{\Omega \vdash \pi_i(w) : \tau_i} \quad \frac{\Omega(w) = \mathsf{bool} \quad \Omega \vdash e_{\mathsf{t}} : \tau \quad \Omega \vdash e_{\mathsf{f}} : \tau}{\Omega \vdash \mathsf{if}\ w\ \mathsf{then}\ e_{\mathsf{t}}\ \mathsf{else}\ e_{\mathsf{f}} : \tau}
$$

$$
\frac{\Omega(w) = \{\tau\} \quad \Omega, x{:}\tau \vdash e : \{\tau'\}}{\Omega \vdash \bigcup\{e \mid x \in w\} : \{\tau'\}} \quad \frac{\Omega(w) = \{\tau\} \quad \Omega, x{:}\tau \vdash e : \mathsf{int}}{\Omega \vdash \sum\{e \mid x \in w\} : \{\tau\}}
$$

Figure 6. Well-formed normalized NRC expressions

$$
\frac{}{\Psi \vdash_{\mathsf{con}} i : \mathsf{int}} \quad \frac{}{\Psi \vdash_{\mathsf{con}} b : \mathsf{bool}} \quad \frac{}{\Psi \vdash_{\mathsf{con}} (l_1, l_2) : \Psi(l_1) \times \Psi(l_2)}
$$

$$
\frac{\tau = \Psi(l_1) = \cdots = \Psi(l_n)}{\Psi \vdash_{\mathsf{con}} \{l_1 : m_1, \ldots, l_n : m_n\} : \{\tau\}} \quad \frac{}{\cdot : \cdot} \quad \frac{\sigma : \Psi \quad \Psi \vdash_{\mathsf{con}} k : \tau}{\sigma, l \mapsto k : \Psi, l : \tau}
$$

Figure 7. Store and constructor well-formedness

The operational semantics is illustrated on the Examples 1–2 in Figure 1; here, the labels $r, r_1, \ldots, s, \ldots$ uniquely identify each part of the input tables R, S and the labels on the results reflect one possible labeling that is consistent with examples given later.

2.5 Type system for A-normalized expressions

We define typing rules for (normalized) NRC expressions as shown in Figure 6. We use standard *contexts* $\Gamma ::= \cdot \mid \Gamma, x{:}\tau$ mapping variables to types and *store types* Ψ of the form $\Psi ::= \cdot \mid \Psi, l{:}\tau$. For brevity, we write Ω for a pair Ψ, Γ and $\Omega(w)$ for $\Psi(l)$ if $l = w$ or $\Gamma(x)$ if $w = x$ respectively. The judgment $\Psi, \Gamma \vdash e : \tau$ means that given store type Ψ and context Γ, expression e has type τ.

The well-formedness judgment for stores is $\sigma : \Psi$, or "σ has store type Ψ". This judgment is defined in Figure 7, using an auxiliary judgment $\Psi \vdash_{\mathsf{con}} k : \tau$, meaning "in stores of type Ψ, constructor k has type τ". The well-formedness judgment for environments $\gamma : \textit{Var} \to \textit{Lab}$ is $\Psi \vdash \gamma : \Gamma$, or "in a store with type Ψ, environment γ matches context Γ". The rules are as follows:

$$
\frac{}{\Psi \vdash \cdot : \cdot} \quad \frac{\Psi \vdash \gamma : \Gamma \quad \Psi(l) = \tau}{\Psi \vdash \gamma, x \mapsto l : \Gamma, x \mapsto \tau}
$$

We sometimes combine the judgments and write $\Psi \vdash \sigma, \gamma : \Gamma$ to indicate $\sigma : \Psi$ and $\Psi \vdash \gamma : \Gamma$. The operational semantics is sound with respect to the store typing rules:

Theorem 1. *Suppose $\Psi \vdash e : \tau$ and $\sigma : \Psi$. Then if $\sigma, l \Leftarrow e \Downarrow \sigma'$ then there exists Ψ' such that $\Psi'(l) = \tau$ and $\sigma' : \Psi'$.*

2.6 Correctness of operational semantics

To show the correctness of the operational semantics relative to the denotational semantics, we need to translate from stores and labels to values. We define the functions $\sigma \uparrow_\tau l$ by induction on types as

follows:

$$\begin{aligned}
\sigma \uparrow_{\mathsf{int}} l &= \sigma(l) \\
\sigma \uparrow_{\mathsf{bool}} l &= \sigma(l) \\
\sigma \uparrow_{\tau_1 \times \tau_2} l &= (\sigma \uparrow_{\tau_1} \pi_1(\sigma(l)), \sigma \uparrow_{\tau_2} \pi_2(\sigma(l))) \\
\sigma \uparrow_{\{\tau\}} l &= \{\sigma \uparrow_{\tau} l' \mid l' \in \sigma(l)\}
\end{aligned}$$

We also define $\sigma \uparrow_\Gamma \gamma$ pointwise, so that $(\sigma \uparrow_\Gamma \gamma)(x) = \sigma \uparrow_{\Gamma(x)} \gamma(x)$. We can easily show that:

Proposition 2. *If $\sigma : \Psi$ and $l : \tau \in \Psi$ then $\sigma \uparrow_\tau l \in [\![\tau]\!]$. Moreover, if $\Psi \vdash \gamma : \Gamma$ then $\sigma \uparrow_\Gamma \gamma \in [\![\Gamma]\!]$.*

The correctness of the operational semantics can then be established by induction on the structure of derivations:

Proposition 3. *Suppose that $\Gamma \vdash e : \tau$ and $\Psi \vdash \sigma, \gamma : \Gamma$. Then there exists σ' such that $\sigma, l \Leftarrow \gamma(e) \Downarrow \sigma'$. Moreover, for any such σ', $[\![e]\!](\sigma \uparrow_\Gamma \gamma) = \sigma' \uparrow_\tau l$.*

3. Traced evaluation and adaptation

We now consider *traces* which are intended to capture the "execution history" of a query in a form that is itself suitable for querying. We define *traces T* using the *terms* introduced earlier as follows:

$$\begin{aligned}
T &::= l \leftarrow t \mid l \leftarrow \mathsf{proj}_i(l', l'') \mid \mathsf{cond}_l(l', b, T)_{e_1}^{e_2} \mid T_1; T_2 \\
&\quad \mid l \leftarrow \mathsf{sum}(l', \Theta)_{x.e} \mid l \leftarrow \mathsf{comp}(l', \Theta)_{x.e} \\
\Theta &::= \{[l_1]T_1 : m_1, \dots, [l_n]T_n : m_n\}
\end{aligned}$$

Terms, introduced above, describe single computation steps. Labeled trace collections Θ are multisets of labeled traces $[l]T$. *Assignment traces* $l \leftarrow t$ record that a new label l was created and assigned the value described by trace term t. *Projection traces* $l \leftarrow \mathsf{proj}_i(l', l'')$ record that l was created and assigned the value at l'', by projecting the i-th component of pair l'. *Sequential composition traces* $T_1; T_2$ indicate that T_1 was performed first followed by T_2. *Conditional traces* $\mathsf{cond}_l(l', b, T)_{e_1}^{e_2}$ record that a conditional expression tested l', found it equal to boolean b, and then performed trace T that writes to l. In addition, conditional traces record the alternative expressions e_1 and e_2 corresponding to the true and false branches. *Comprehension traces* $l \leftarrow \mathsf{comp}(l', \Theta)_{x.e}$ record that l was created by performing a comprehension over the set at l', with subtraces Θ describing the iterations; the expression $x.e$ records the body of the comprehension with its bound variable x. Sum traces $l \leftarrow \mathsf{sum}(l, \Theta)_{x.e}$ are similar.

When the expressions $e_1, e_2, x.e$ in conditional or comprehension traces are irrelevant to the discussion we often omit them for brevity, e.g. writing $\mathsf{cond}_l(l', b, T)$ or $\mathsf{comp}(l, \Theta)$.

We define the result label of a trace as follows:

$$\begin{aligned}
\mathrm{out}(l \leftarrow t) &= l \\
\mathrm{out}(T_1; T_2) &= \mathrm{out}(T_2) \\
\mathrm{out}(\mathsf{cond}_l(l', b, T)_{e_1}^{e_2}) &= l \\
\mathrm{out}(l \leftarrow \mathsf{proj}_i(l', l'')) &= l \\
\mathrm{out}(l \leftarrow \mathsf{comp}(l', \Theta)_{x.e}) &= l \\
\mathrm{out}(l \leftarrow \mathsf{sum}(l', \Theta)_{x.e}) &= l
\end{aligned}$$

We define the input labels of a labeled trace set Θ as $\mathrm{in}^\star(\Theta) = \{l : m \mid [l]T : m \in \Theta\}$. Similarly, the result labels of Θ are defined as $\mathrm{out}^\star(\Theta) = \{\mathrm{out}(T) : m \mid [l]T : m \in \Theta\}$. Note that we treat both as multisets.

3.1 Traced operational semantics

We now define *traced evaluation*, a refinement of the operational semantics in Section 2.4. The rules for traced evaluation are shown in Figure 8. There are two judgments: $\sigma, l \Leftarrow e \Downarrow \sigma', T$, meaning

$$\frac{}{\sigma, l \Leftarrow t \Downarrow \sigma[l := \mathrm{op}(t, \sigma)], l \leftarrow t}$$

$$\frac{\sigma, l' \Leftarrow e_1 \Downarrow \sigma_1, T_1 \quad \sigma, l \Leftarrow e_2[l'/x] \Downarrow \sigma_2, T_2}{\sigma, l \Leftarrow \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow \sigma_2, T_1; T_2}\ l'\ \text{fresh}$$

$$\frac{\sigma(l') = b \quad \sigma, l \Leftarrow e_b \Downarrow \sigma', T}{\sigma, l \Leftarrow \mathsf{if}\ l'\ \mathsf{then}\ e_\mathsf{t}\ \mathsf{else}\ e_\mathsf{f} \Downarrow \sigma', \mathsf{cond}_l(l', b, T)_{e_\mathsf{t}}^{e_\mathsf{f}}}$$

$$\frac{\sigma(l') = (l_1, l_2)}{\sigma, l \Leftarrow \pi_i l' \Downarrow \sigma[l := \sigma(l_i)], l \leftarrow \mathsf{proj}_i(l', l_i)}$$

$$\frac{\sigma, x \in \sigma(l'), e \Downarrow^\star \sigma', L', \Theta}{\sigma, l \Leftarrow \bigcup\{e \mid x \in l'\} \Downarrow \sigma'[l := \bigsqcup \sigma'[L']], l \leftarrow \mathsf{comp}(l', \Theta)_{x.e}}$$

$$\frac{\sigma, x \in \sigma(l'), e \Downarrow^\star \sigma', L', \Theta}{\sigma, l \Leftarrow \sum\{e \mid x \in l'\} \Downarrow \sigma'[l := \sum \sigma'[L']], l \leftarrow \mathsf{sum}(l', \Theta)_{x.e}}$$

$$\frac{\sigma, l' \Leftarrow e[l/x] \Downarrow \sigma', T \quad l'\ \text{fresh}}{\sigma, x \in \emptyset, e \Downarrow^\star \sigma, \emptyset, \emptyset \qquad \sigma, x \in \{l : m\}, e \Downarrow^\star \sigma', \{l' : m\}, \{[l]T : m\}}$$

$$\frac{\sigma, x \in L_1, e \Downarrow^\star \sigma_1, L_1', \Theta_1 \quad \sigma, x \in L_2, e \Downarrow^\star \sigma_2, L_2', \Theta_2}{\sigma, x \in L_1 \oplus L_2, e \Downarrow^\star \sigma_1 \uplus_\sigma \sigma_2, L_1' \oplus L_2', \Theta_1 \oplus \Theta_2}$$

Figure 8. Traced evaluation

"Starting in store σ, evaluating e and storing the result at l yields store σ' and trace T", and $\sigma, x \in L, e \Downarrow^\star \sigma', \vec{L'}, \Theta$, meaning "Starting in store σ, evaluating e with x bound to each label in L in turn yields store σ', result labels L' and labeled traces Θ".

Each operational semantics rule relates a different expression form to its trace form. Thus, traces can be viewed as explaining the dynamic execution history of the expression. (We will make this precise in Section 4). In particular, terms t are translated to assignment traces. Let-expressions are translated to sequential compositions of traces. For these expressions, it would be superfluous to record additional information such as the values of the inputs and outputs, since this can be recovered from the input store and the trace (as we shall see below). However, more detailed trace information is needed for some expressions, such as projections, conditionals, comprehensions, and sums. Their traces record some expression annotations and some information about the structure of the input store. Conditionals record the boolean value of the conditional test as well as both branches of the conditional; comprehensions and sums record the labels and subtraces of the elements of the input set as well as the body of the comprehension. This information is necessary to obtain the fidelity property (Section 4) and to ensure that we can extract other forms of provenance from traces (Section 5).

Example 3 Figure 9 shows one possible trace resulting from normalizing and running query Q_1 from Example 1 on the data in Figure 1. Similarly, Figure 10 shows a possible trace of the grouping-aggregation query Q_2 from Example 2. Since the example queries use record syntax, we use terms such as $(\vec{A} : \vec{l})$ and traces $l \leftarrow \mathsf{proj}_A(l', l'')$ for record construction and field projection. These operations are natural generalizations of pair terms and projection traces. For brevity, the examples omit expression annotations.

We will need the following property:

Lemma 1. *If $\sigma, l \Leftarrow e \Downarrow \sigma', T$ then $\mathrm{out}(T) = l$.*

Proof. Easy induction on derivations. ☐

3.2 Adaptive semantics

We also introduce an *adaptive semantics* that adapts traces to changes in the input. Similarly to change-propagation in AFL (Acar et al. 2006), we can use the adaptive semantics to "recompute" an expression when the input is changed, and to adapt the trace to be

```
l <- comp(r,{
  [r1] x11 <- proj_C(r1,r13); x1 <- comp(s,{
    [s1] x111 <- proj_C(s1,s11); x112 <- x11 = x111;
         cond(x112,f,x113 <- {}),
    [s2] x121 <- proj_C(s2,s21); x122 <- x11 = x121;
         cond(x122,f,x123 <- {}),
    [s3] x131 <- proj_C(s3,s31); x132 <- x11 = x131;
         cond(x132,t,l11 <- proj_A(r1,r11);
                     l12 <- proj_B(r1,r12);
                     l13 <- proj_D(s3,s32);
                     l1 <- (A:l11,B:l12,D:l13);
                     x136 <- {l1})}),
  [r2] x21 <- proj_C(r2,r23); x2 <- comp(s,{
    [s1] x211 <- proj_C(s1,s11); x212 <- x21 = x211;
         cond(x212,f,x213 <- {}),
    [s2] x221 <- proj_C(s2,s21); x222 <- x21 = x221;
         cond(x222,f,x223 <- {}),
    [s3] x231 <- proj_C(s3,s31); x232 <- x21 = x231;
         cond(x232,t,l21 <- proj_A(r2,r21);
                     l22 <- proj_B(r2,r22);
                     l23 <- proj_D(s3,s32);
                     l2 <- (A:l21,B:l22,D:l23);
                     x126 <- {l2})}),
  [r3] x31 <- proj_C(r3,r33); x3 <- comp(s,{
    [s1] x311 <- proj_C(s1,s11); x312 <- x31 = x311;
         cond(x312,f,x313 <- {}),
    [s2] x321 <- proj_C(s2,s21); x322 <- x31 = x321;
         cond(x322,f,x323 <- {}),
    [s3] x331 <- proj_C(s3,s31); x332 <- x31 = x331;
         cond(x332,f,x333 <- {})})})
```

Figure 9. Example trace for Q_1

```
l11' <- 42;  x1 <- 2;
l12' <- sum(s,{
  [s1] x11 <- proj_C(s1,s11); x12 <- x11 = x1;
       cond(x12,t, x13 <- proj_D(s1,s12)),
  [s2] x21 <- proj_C(s2,s21); x22 <- x21 = x1;
       cond(x22,t, x23 <- proj_D(s2,s22)),
  [s3] x31 <- proj_C(s3,s31); x32 <- x31 = x1;
       cond(x32,f, x33 <- 0)});
l1' <- (C:l11',D:l12'); x <- {l1'}; y12 <- 4;
y <- comp(r,{
  [r1] y11 <- proj_C(r1,r13); y12 <- y11 = y1;
       cond(y12,f, y13 <- {}),
  [r2] y21 <- proj_C(r2,r21); y22 <- y21 = y1;
       cond(y22,f,y23 <- {}),
  [r3] y31 <- proj_C(r3,r31); y32 <- y31 = y1;
       cond(y32,t,l21' <- proj_B(r3,r32);
                  l22' <- proj_A(r3,r31);
                  l2' <- (C:l21',D:l22')
                  y33 <- {l2'})});
l' <- x U y
```

Figure 10. Trace example for query Q_2

$$\sigma, l \leftarrow t \curvearrowright \sigma[l := \mathrm{op}(t,\sigma)], l \leftarrow t$$

$$\frac{\sigma, T_1 \curvearrowright \sigma', T_1' \quad \sigma', T_2 \curvearrowright \sigma'', T_2'}{\sigma, T_1; T_2 \curvearrowright \sigma'', T_1'; T_2'}$$

$$\frac{\sigma(l') = (l_1', l_2')}{\sigma, l \leftarrow \mathsf{proj}_i(l', l_i) \curvearrowright \sigma[l := l_i], l \leftarrow \mathsf{proj}_i(l', l_i')}$$

$$\frac{b' = \sigma(l') \neq b \quad \sigma, l \Leftarrow e_{b'} \Downarrow \sigma', T'}{\sigma, \mathsf{cond}_l(l', b, T)_{e_1}^{e_2} \curvearrowright \sigma', \mathsf{cond}_l(l', b', T')_{e_1}^{e_2}}$$

$$\frac{\sigma(l') = b \quad \sigma, T \curvearrowright \sigma', T' \quad l = \mathrm{out}(T')}{\sigma, \mathsf{cond}_l(l', b, T)_{e_1}^{e_2} \curvearrowright \sigma', \mathsf{cond}_l(l', b, T')_{e_1}^{e_2}}$$

$$\frac{\sigma, x \in \sigma(l'), e, \Theta \curvearrowright^\star \sigma', L', \Theta'}{\sigma, l \leftarrow \mathsf{comp}(l', \Theta)_{x.e} \curvearrowright \sigma'[l := \bigsqcup \sigma'[L']], l \leftarrow \mathsf{comp}(l', \Theta')_{x.e}}$$

$$\frac{\sigma, x \in \sigma(l'), e, \Theta \curvearrowright^\star \sigma', L', \Theta'}{\sigma, l \leftarrow \mathsf{sum}(l', \Theta)_{x.e} \curvearrowright \sigma'[l := \sum \sigma'[L']], l \leftarrow \mathsf{sum}(l', \Theta')_{x.e}}$$

$$\sigma, x \in \emptyset, e, \Theta \curvearrowright^\star \sigma, \emptyset, \emptyset$$

$$\frac{[l]T \in \Theta \quad \sigma, T \curvearrowright \sigma', T'}{\sigma, x \in \{l : m\}, e, \Theta \curvearrowright^\star \sigma', \{\mathrm{out}(T') : m\}, \{[l]T' : m\}}$$

$$\frac{l \notin \mathrm{in}^\star(\Theta) \quad l' \text{ fresh} \quad \sigma, l' \Leftarrow e[l/x] \Downarrow \sigma', T'}{\sigma, x \in \{l : m\}, e, \Theta \curvearrowright^\star \sigma', \{l' : m\}, \{[l]T' : m\}}$$

$$\frac{\sigma, x \in L_1, e, \Theta \curvearrowright^\star \sigma_1, L_1', \Theta_1 \quad \sigma, x \in L_2, e, \Theta \curvearrowright^\star \sigma_2, L_2', \Theta_2}{\sigma, x \in L_1 \oplus L_2, e, \Theta \curvearrowright^\star \sigma_1 \uplus_\sigma \sigma_2, L_1' \oplus L_2', \Theta_1 \oplus \Theta_2}$$

Figure 11. Trace adaptation semantics

consistent with the new input and output. However, unlike in AFL, our goal here is not to efficiently recompute results, but rather to characterize how traces "represent" or "explain" computations. We believe efficient techniques for recomputing database queries could also be developed using similar ideas, but view this as beyond the scope of this paper.

We define the adaptive semantics rules in Figure 11. Following the familiar pattern established by the operational semantics, we use two judgments: $\sigma, T \curvearrowright \sigma', T'$, or "Recomputing T on σ yields result σ' and new trace T'", and $\sigma, x \in L, e, \Theta \curvearrowright^\star \sigma', L', \Theta'$, or "Reiterating e on σ for each $x \in L$ with cached traces Θ yields result σ', result labels L', and new trace Θ'".

Many of the basic trace steps have straightforward adaptation rules. For example, the rule for traces $l \leftarrow t$ simply recomputes the result using the values of the input labels in the current store. For projection, we recompute the operation and discard the cached labels. Adaptation for sequential composition is also straightforward. For conditional traces, there are two rules. If the boolean value of the label is the same as that recorded in the trace, then we proceed by re-using the subtrace. Otherwise, we need to fall back on the trace semantics to compute the other branch.

The rules for comprehension and summation traces make use of the iteration adaptation judgment. In each case, we traverse the current store value of l_0. For each label l in this set, we re-compute the body of the comprehension, re-using a trace $[l]T$ if present in Θ, otherwise evaluating $e[l/x]$ in the traced semantics. The iterative judgments return a new labeled trace set Θ and its return labels L'. Note that trace adaptation ignores the multiplicity of cached traces. When we re-use a cached trace $[l]T$ on a label l with multiplicity m, we simply rerun the trace and use m as the multiplicity of the result label and new trace.

4. Metatheory

We now investigate the metatheoretic properties of the traced evaluation and trace adaptation semantics.

$$\dfrac{\sigma(l) = \mathrm{op}(t, \sigma)}{\sigma \models l \leftarrow t} \qquad \dfrac{\sigma(l') = (l_1, l_2) \quad \sigma(l) = \sigma(l_i)}{\sigma \models l \leftarrow \mathsf{proj}_i(l', l_i)}$$

$$\dfrac{\sigma \models T_1 \quad \sigma \models T_2}{\sigma \models T_1 ; T_2} \qquad \dfrac{\sigma(l') = b \quad \sigma \models T \quad \mathrm{out}(T) = l}{\sigma \models \mathsf{cond}_l(l', b, T)^{e_2}_{e_1}}$$

$$\dfrac{\sigma(l') = \mathrm{in}^{\star}(\Theta) \quad \sigma \models^{\star} \Theta \quad \sigma(l) = \bigsqcup \sigma[\mathrm{out}^{\star}(\Theta)]}{\sigma \models l \leftarrow \mathsf{comp}(l', \Theta)_{x.e}}$$

$$\dfrac{\sigma(l') = \mathrm{in}^{\star}(\Theta) \quad \sigma \models^{\star} \Theta \quad \sigma(l) = \sum \sigma[\mathrm{out}^{\star}(\Theta)]}{\sigma \models l \leftarrow \mathsf{sum}(l', \Theta)_{x.e}}$$

$$\dfrac{}{\sigma \models^{\star} \emptyset} \qquad \dfrac{\sigma \models^{\star} \Theta_1 \quad \sigma \models^{\star} \Theta_2}{\sigma \models^{\star} \Theta_1 \oplus \Theta_2} \qquad \dfrac{\sigma \models T}{\sigma \models^{\star} \{[l]T : m\}}$$

Figure 12. Declarative semantics of traces

We first show that the traced semantics correctly implements the operational semantics of NRC expressions, if we ignore traces. This is a straightforward induction in both directions.

Theorem 2. *For any σ, l, e, σ', we have $\sigma, l \Leftarrow e \Downarrow \sigma'$ if and only if $\sigma, l \Leftarrow e \Downarrow \sigma', T$ for some T.*

We now turn to the correctness of the trace semantics. We can view the trace semantics as both evaluating e in a store σ yielding σ' and translating e to a trace T which "explains" the execution of e. What properties should a trace have in order to be a valid explanation? We identify two such properties which help to formalize this intuition. They are called *consistency* and *fidelity*.

Consistency The trace is meant to be an explanation of what happened when e was evaluated on σ. For example, if the trace says that $l \leftarrow l_1 + l_2$ but $\sigma'(l) \neq \sigma'(l_1) + \sigma'(l_2)$ then this is inconsistent with the real execution. Also, if the trace contains $\mathsf{cond}_l(l', \mathsf{f}, T)^{e_2}_{e_1}$, but l' actually evaluated to t in the evaluation of e, then the trace is inconsistent with the actual execution. As a third example, if the trace contains $l' \leftarrow \mathsf{comp}(l, \{[l_1]T_1, [l_2]T_2\})_{x.e}$ whereas $\sigma(l) = \{l_2, l_3\}$ then the trace is inconsistent because it does not correctly show the behavior of the comprehension over l.

To formalize this notion of *consistency*, observe that we can view a trace declaratively as a collection of statements about the values in the store. We define a judgment $\sigma \models T$, meaning "T is satisfied in store σ". We also employ an auxiliary judgment $\sigma \models^{\star} \Theta$, meaning "Each trace in Θ is satisfied in store σ". The satisfiability relation is defined in Figure 12.

Theorem 3 (Consistency). *If $\sigma, l \Leftarrow e \Downarrow \sigma', T$ then $\sigma' \models T$.*

Fidelity Consistency is a necessary, but not sufficient, requirement for traces to be "explanations". It tells us that the trace records valid information about the results of an execution. However, this is not enough, in itself, to say that the trace really "explains" the execution, because a consistent trace might not tell us what might have happened in other possible executions. To see why, consider a simple expression if l_y then $l_x + l_z$ else l_z run against input store $[l_x = 42, l_y = \mathsf{t}, l_z = 5]$. Consider the traces, $T_1 = l \leftarrow l_x + l_z$ and $T_2 = l \leftarrow 47$. Both of these traces are consistent, but neither really "explain" what actually happened. Saying that $l \leftarrow l_x + l_z$ or $l \leftarrow 47$ is enough to know what the result value was in the actual run, but not what the result would have been under all conditions. The dependence on l_x is lost in T_2. If we rerun T_1 with a different input store $l_x = 37$, then T_1 will correctly return 42 while T_2 will still return 47. Moreover, the dependences on l_y are lost in both: changing l_y to f invalidates both traces. Instead, the trace $T_3 = \mathsf{cond}_l(l_y, \mathsf{t}, l \leftarrow l_x + l_z)^{l_z}_{l_x + l_z}$ records enough information

to recompute the result under *any* (reasonable) change to the input store.

We call traces *faithful* to e if they record enough information to recompute e when the input store changes. We first consider a property called *partial fidelity*. Partial fidelity tells us that the trace adaptation semantics is partially correct with respect to the traced evaluation semantics. That is, if T was obtained by running e on σ_1 and we can successfully adapt T to a new input σ_2 to obtain σ'_2 and T', then we know that σ'_2 and T' could also have been obtained by traced evaluation from σ_2 "from scratch".

We first need some lemmas:

Lemma 2. *If $[l]T \in \Theta$ and $\sigma, x \in L, e \Downarrow^{\star} \sigma', L', \Theta$ then for some σ'' we have $\sigma, \mathrm{out}(T) \Leftarrow e[l/x] \Downarrow \sigma'', T$.*

Proof. Induction on the structure of $\sigma, x \in L, e \Downarrow^{\star} \sigma', L', \Theta$.

- The case where $\Theta = \emptyset$ is vacuous since $[l]T \in \Theta$.
- Suppose the derivation is of the form

$$\dfrac{\sigma, x \in L_1, e \Downarrow^{\star} \sigma_1, L'_1, \Theta_1 \quad \sigma, x \in L_2, e \Downarrow^{\star} \sigma_2, L'_2, \Theta_2}{\sigma, x \in L_1 \cup L_2, e \Downarrow^{\star} \sigma_1 \uplus_{\sigma} \sigma_2, L'_1 \cup L'_2, \Theta_1 \oplus \Theta_2}$$

Then either $[l]T \in \Theta_1$ or $[l]T \in \Theta_2$; the cases are symmetric. In either case, the induction hypothesis applies and we have $\sigma, \mathrm{out}(T) \Leftarrow e[l/x] \Downarrow \sigma_i, T$ as desired.
- Suppose the derivation is of the form

$$\dfrac{\sigma, l' \Leftarrow e[l/x] \Downarrow \sigma', T}{\sigma, x \in \{l : m\}, e \Downarrow^{\star} \sigma', \{l' : m\}, \{[l]T : m\}}$$

Then the subderivation $\sigma, l' \Leftarrow e[l/x] \Downarrow \sigma', T$ is the desired conclusion. $\qquad \square$

Lemma 3. *If $[l]T \in \Theta$ and $\Psi \vdash \tau \triangleright \Theta \triangleright \tau'$ then we have $\Psi, l{:}\tau \vdash T \triangleright \mathrm{out}(T) : \tau'$.*

Proof. Straightforward induction similar to Lemma 2. $\qquad \square$

Lemma 4. *If $\sigma, T \curvearrowright \sigma', T'$ then $\mathrm{out}(T) = \mathrm{out}(T')$.*

Proof. Straightforward induction on derivations. $\qquad \square$

Theorem 4 (Partial fidelity). *Let $\sigma_1, \sigma'_1, \sigma_2, \sigma'_2, T, T', \Theta, \Theta'$ be given.*

1. *If $\sigma_1, l \Leftarrow e \Downarrow \sigma'_1, T$ and $\sigma_2, T \curvearrowright \sigma'_2, T'$ then $\sigma_2, l \Leftarrow e \Downarrow \sigma'_2, T'$.*
2. *If $\sigma_1, x \in L_1, e \Downarrow^{\star} \sigma'_1, L'_1, \Theta$ and $\sigma_2, L_2 \in e, \Theta, \sigma'_2 \curvearrowright^{\star} L'_2, \Theta'$, then $\sigma_2, x \in L_2, e \Downarrow^{\star} \sigma'_2, L'_2, \Theta'$*

Proof. Induction on the structure of the second derivation, with inversion on the first derivation. Lemma 2 is needed in part (2) to deal with the adaptation case where $[l]T \in \Theta$ holds.

For part 1, the cases are as follows:

- If the second derivation is of the form

$$\dfrac{}{\sigma_2, l \leftarrow t \curvearrowright \sigma_2[l := \mathrm{op}(t, \sigma_2)], l \leftarrow t}$$

then the first must be of the form

$$\dfrac{}{\sigma_1, l \leftarrow t \Downarrow \sigma_1[l := \mathrm{op}(t, \sigma_1)], l \leftarrow t}$$

and so we can immediately conclude

$$\dfrac{}{\sigma_2, l \leftarrow t \Downarrow \sigma_2[l := \mathrm{op}(t, \sigma_2)], l \leftarrow t}$$

- If the second derivation is of the form

$$\frac{\sigma_2(l') = (l_1', l_2')}{\sigma_2, l \leftarrow \mathsf{proj}_i(l', l_i) \curvearrowright \sigma_2[l := \sigma_2(l_i')], l \leftarrow \mathsf{proj}_i(l', l_i')}$$

then the first derivation is of the form

$$\frac{\sigma_1(l') = (l_1, l_2)}{\sigma_1, l \Leftarrow \pi_i(l') \Downarrow \sigma_1[l := \sigma_1(l_i)], l \leftarrow \mathsf{proj}_i(l', l_i)}$$

and so we can immediately conclude

$$\frac{\sigma_2(l') = (l_1', l_2')}{\sigma_2, l \Leftarrow \pi_i(l') \Downarrow \sigma_2[l := \sigma_2(l_i')], l \leftarrow \mathsf{proj}_i(l', l_i')}$$

- If the second derivation is of the form

$$\frac{\sigma_2, T_{11} \curvearrowright \sigma_2', T_{21} \quad \sigma_2', T_{12} \curvearrowright \sigma_2'', T_{22}}{\sigma_2, T_{11}; T_{12} \curvearrowright \sigma_2'', T_{21}; T_{22}}$$

then the first derivation must be of the form

$$\frac{\sigma_1, l' \Leftarrow e_1 \Downarrow \sigma_1', T_{11} \quad \sigma_1', l \Leftarrow e_2[l'/x] \Downarrow \sigma_1'', T_{12}}{\sigma_1, l \Leftarrow \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow \sigma_1'', T_{11}; T_{12}}$$

Then by induction we have $\sigma_2, l' \Leftarrow e_1 \Downarrow \sigma_2', T_{21}$ and $\sigma_2', l \Leftarrow e_2[l/x] \Downarrow \sigma_2'', T_{22}$, so can conclude

$$\frac{\sigma_2, l' \Leftarrow e_1 \Downarrow \sigma_2', T_{21} \quad \sigma_2', l \Leftarrow e_2[l'/x] \Downarrow \sigma_2'', T_{22}}{\sigma_2, l \Leftarrow \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow \sigma_2'', T_{21}; T_{22}}$$

- If the second derivation is of the form

$$\frac{\sigma_2(l) = b \quad \sigma_2, T_1 \curvearrowright \sigma_2', T_2}{\sigma_2, \mathsf{cond}_l(l', b, T_1)^{e_f}_{e_t} \curvearrowright \sigma_2', \mathsf{cond}_l(l', b, T_2)^{e_f}_{e_t}}$$

then the first derivation must be of the form

$$\frac{\sigma_1(l') = b \quad \sigma_1, l \Leftarrow e_b \Downarrow \sigma_1', T_1}{\sigma_1, l \Leftarrow \mathsf{if}\ l'\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \Downarrow \sigma_1', \mathsf{cond}_l(l', b, T_1)^{e_f}_{e_t}}$$

We proceed by induction, obtaining $\sigma_2, l \Leftarrow e_b \Downarrow \sigma_2', T_2$ and concluding

$$\frac{\sigma_2(l) = b \quad \sigma_2, l \Leftarrow e_b \Downarrow \sigma_2', T_2}{\sigma_2, l \Leftarrow \mathsf{if}\ l'\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \Downarrow \sigma_2', \mathsf{cond}_l(l', b, T_2)^{e_f}_{e_t}}$$

- If the second derivation is of the form:

$$\frac{b \neq \sigma_2(l) = b' \quad \sigma_2, l \Leftarrow e_{b'} \Downarrow \sigma_2', T_2}{\sigma_2, \mathsf{cond}_l(l', b, T_1)^{e_f}_{e_t} \curvearrowright \sigma_2', \mathsf{cond}_l(l', b, T_2)^{e_f}_{e_t}}$$

then again the first derivation must be of the form

$$\frac{\sigma_1(l') = b \quad \sigma_1, l \Leftarrow e_b \Downarrow \sigma_1', T_1}{\sigma_1, l \Leftarrow \mathsf{if}\ l'\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \Downarrow \sigma_1', \mathsf{cond}_l(l', b, T_1)^{e_f}_{e_t}}$$

amd we may immediately conclude:

$$\frac{\sigma_2(l) = b' \quad \sigma_2, l \Leftarrow e_{b'} \Downarrow \sigma_2', T_2}{\sigma_2, l \Leftarrow \mathsf{if}\ l'\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \Downarrow \sigma_2', \mathsf{cond}_l(l', b', T_2)^{e_f}_{e_t}}$$

- If the second derivation is of the form

$$\frac{\sigma_2, x \in \sigma_2(l'), e, \Theta_1 \curvearrowright^\star \sigma_2', L_2, \Theta_2}{\sigma_2, l \leftarrow \mathsf{comp}(l', \Theta_1)_{x.e} \curvearrowright \sigma_2'[l := \bigsqcup \sigma_2'[L_2]], l \leftarrow \mathsf{comp}(l', \Theta_2)_{x.e}}$$

then the first derivation must be of the form

$$\frac{\sigma_1, x \in \sigma_1(l'), e \Downarrow^\star \sigma_1', L_1, \Theta_1}{\sigma_1, l \Leftarrow \bigcup\{e \mid x \in l'\} \Downarrow \sigma_1'[l := \bigsqcup \sigma'[L_1]], l \leftarrow \mathsf{comp}(l', \Theta_1)_{x.e}}$$

By induction hypothesis (2), we have that $\sigma_2, x \in \sigma_2(l'), e \Downarrow^\star \sigma_2', L_2, \Theta_2$ holds, so can conclude:

$$\frac{\sigma_2, x \in \sigma_2(l'), e \Downarrow^\star \sigma_2', L_2, \Theta_2}{\sigma_2, l \Leftarrow \bigcup\{e \mid x \in l'\} \Downarrow \sigma_2'[l := \bigsqcup \sigma'[L_2]], l \leftarrow \mathsf{comp}(l', \Theta_2)_{x.e}}$$

- If the second derivation is of the form

$$\frac{\sigma_2, x \in \sigma_2(l'), e, \Theta_1 \curvearrowright^\star \sigma_2', L_2, \Theta_2}{\sigma_2, l \leftarrow \mathsf{sum}(l', \Theta_1)_{x.e} \curvearrowright \sigma_2'[l := \sum \sigma_2'[L_2]], l \leftarrow \mathsf{sum}(l', \Theta_2)_{x.e}}$$

the reasoning is similar to the previous case.

For part (2), the proof is by induction on the second derivation:

- If the derivation is of the form:

$$\overline{\sigma_2, x \in \emptyset, e, \Theta_1 \curvearrowright^\star \sigma_2, \emptyset, \emptyset}$$

then we can immediately conclude

$$\overline{\sigma_2, x \in \emptyset, e \Downarrow^\star \sigma_2, \emptyset, \emptyset}$$

- If the derivation is of the form:

$$\frac{\sigma_2, x \in L_{21}, e, \Theta_1 \curvearrowright^\star \sigma_{21}, L_{21}', \Theta_{21}}{\sigma_2, x \in L_{22}, e, \Theta_1 \curvearrowright^\star \sigma_{22}, L_{22}', \Theta_{22}}{\sigma_2, x \in L_{21} \cup L_{22}, e, \Theta_1 \curvearrowright^\star \sigma_{21} \uplus_{\sigma_2} \sigma_{22}, L_{21}' \cup L_{22}', \Theta_{21} \cup \Theta_{22}}$$

then we proceed by induction, concluding:

$$\frac{\sigma_2, x \in L_{21}, e \Downarrow^\star \sigma_{21}, L_{21}', \Theta_{21}}{\sigma_2, x \in L_{22}, e \Downarrow^\star \sigma_{22}, L_{22}', \Theta_{22}}{\sigma_2, x \in L_{21} \cup L_{22}, e \Downarrow^\star \sigma_{21} \uplus_{\sigma_2} \sigma_{22}, L_{21}' \cup L_{22}', \Theta_{21} \cup \Theta_{22}}$$

- If the derivation is of the form

$$\frac{l \notin \mathrm{in}^\star(\Theta_1) \quad l'\ \text{fresh} \quad \sigma_2, l \Leftarrow e[l/x] \Downarrow \sigma_2', T_2}{\sigma_2, x \in \{l : m\}, e, \Theta_1 \curvearrowright^\star \sigma_2', \{l' : m\}, \{[l]T_2 : m\}}$$

then we can immediately conclude:

$$\frac{\sigma_2, l' \Leftarrow e[l/x] \Downarrow \sigma_2', T_2 \quad l'\ \text{fresh}}{\sigma_2, x \in \{l : m\}, e \Downarrow^\star \sigma_2', \{l' : m\}, \{[l]T_2 : m\}}$$

- If the derivation is of the form:

$$\frac{[l]T_1 \in \Theta_1 \quad \sigma_2, T_1 \curvearrowright \sigma_2', T_2}{\sigma_2, x \in \{l : m\}, e, \Theta_1 \curvearrowright^\star \sigma_2', \{\mathrm{out}(T_2) : m\}, \{[l]T_2 : m\}}$$

then observe that $\mathrm{out}(T_1) = \mathrm{out}(T_2)$ by Lemma 4. Moreover, by Lemma 2, we have $\sigma_1, \mathrm{out}(T_1) \Leftarrow e[l/x] \Downarrow \sigma_1'', T_1$, so by induction we have $\sigma_2, \mathrm{out}(T_1) \Leftarrow e[l/x] \Downarrow \sigma_2', T_2$, and we can conclude

$$\frac{\sigma_2, \mathrm{out}(T_1) \Leftarrow e[l/x] \Downarrow \sigma_2', T_2}{\sigma_2, x \in \{l : m\}, e \Downarrow^\star \sigma_2', \{\mathrm{out}(T_2) : m\}, \{[l]T_2 : m\}}$$

$\square$

However, partial fidelity is rather weak since there is no guarantee that T can be adapted to a given σ_2. To formalize and prove total fidelity, we need to be careful about what changed inputs σ_2 we consider. Obviously, σ_2 must be type-compatible with T in some sense; for instance we cannot expect a trace such as $l \leftarrow l_1 + l_2$ to adapt to an input in which $l_1 = \mathsf{t}$. Thus, we need to set up a type system for stores and traces and prove type-soundness for traced evaluation and adaptation.

More subtly, if we have a trace $l \leftarrow t$ that writes to l and we try to evaluate it on a different store that *already defines* l, perhaps at a different type, then the adaptation step may succeed, but the result store may be ill-formed, leading to problems later on. In general, we need to restrict attention to altered stores σ_2 that *preserve the types of labels read by T* and *avoid labels written by T*.

We say that σ *matches* Ψ *avoiding* S (written $\sigma <: \Psi \# S$) if $\sigma : \Psi'$ for some $\Psi' \supseteq \Psi$ with $\mathrm{dom}(\Psi') \cap S = \emptyset$. That is, σ satisfies the type information in Ψ, and may have other labels, but the other labels cannot overlap with S. Moreover, when L is a collection of labels $\{l_1 : m_1, \ldots, l_n : m_n\}$, we sometimes write $L{:}\tau$ as an

$$\frac{\Psi \vdash_{\mathsf{term}} t : \tau}{\Psi \vdash l \leftarrow t \triangleright l : \tau} \qquad \frac{\Psi(l') = \tau_1 \times \tau_2}{\Psi \vdash l \leftarrow \mathsf{proj}_i(l', l_i) \triangleright l : \tau_i}$$

$$\frac{\Psi \vdash T_1 \triangleright l' : \tau' \quad \Psi, l':\tau' \vdash T_2 \triangleright l : \tau}{\Psi \vdash T_1; T_2 \triangleright l : \tau}$$

$$\frac{\Psi(l') = \mathsf{bool} \quad \Psi \vdash T \triangleright l : \tau \quad \Psi \vdash e_{\mathsf{t}} : \tau \quad \Psi \vdash e_{\mathsf{f}} : \tau}{\Psi \vdash \mathsf{cond}_l(l', b, T)_{e_{\mathsf{t}}}^{e_{\mathsf{f}}} \triangleright l : \tau}$$

$$\frac{\Psi(l') = \{\tau'\} \quad \Psi \vdash \tau' \triangleright \Theta \triangleright \{\tau\} \quad \Psi, x:\tau' \vdash e : \{\tau\}}{\Psi \vdash l \leftarrow \mathsf{comp}(l', \Theta)_{x.e} \triangleright l : \{\tau\}}$$

$$\frac{\Psi(l') = \{\tau'\} \quad \Psi \vdash \tau' \triangleright \Theta \triangleright \mathsf{int} \quad \Psi, x:\tau' \vdash e : \mathsf{int}}{\Psi \vdash l \leftarrow \mathsf{sum}(l', \Theta)_{x.e} \triangleright l : \mathsf{int}}$$

$$\frac{}{\Psi \vdash \tau \triangleright \emptyset \triangleright \tau'} \qquad \frac{\Psi, l:\tau \vdash T \triangleright l' : \tau'}{\Psi \vdash \tau \triangleright \{[l]T : m\} \triangleright \tau'}$$

$$\frac{\Psi \vdash \tau \triangleright \Theta_1 \triangleright \tau' \quad \Psi \vdash \tau \triangleright \Theta_2 \triangleright \tau'}{\Psi \vdash \tau \triangleright \Theta_1 \oplus \Theta_2 \triangleright \tau'}$$

Figure 13. Trace well-formedness

abbreviation for $l_1 : \tau, \dots, l_n : \tau$; thus, $\sigma <: \Psi, L:\tau \# S$ stands for $\sigma <: \Psi, l_1:\tau, \dots, l_n:\tau \# S$.

We also need to be careful to avoid making the type system too specific about the labels used internally by T, because these may change when T is adapted. We therefore introduce a typing judgment for traces $\Psi \vdash T \triangleright l : \tau$, meaning "In a store matching type Ψ, trace T produces an output l of type τ." Trace typing does not expose the types of labels created by T for internal use in the rules for let and comprehension. The rules are shown in Figure 13, along with the auxiliary judgment $\Psi \vdash \tau \triangleright \Theta \triangleright \tau'$, meaning "In a store matching type Ψ, the labeled traces Θ operate on inputs of type τ and produce outputs of type τ'".

We now show that for well-formed expressions and input stores, traced evaluation can construct well-formed output stores and traces avoiding any finite set of labels. Here, we need label-avoidance constraints to avoid label conflicts between σ_1 and σ_2 in the $\Downarrow^\star$-rule for $\Theta_1 \oplus \Theta_2$. We also need these constraints later in proving Theorem 7. Next we show traced evaluation is sound, that is, produces well-formed traces and states.

Theorem 5 (Traceability). *Let S be a finite set of labels, and Ψ, e, τ, l, σ be arbitrary.*

1. *If $\Psi \vdash e : \tau$ and $\sigma <: \Psi \# S \cup \{l\}$ then there exists σ', T such that $\sigma, l \Leftarrow e \Downarrow \sigma', T$ and $\sigma' <: \Psi, l:\tau \# S$.*
2. *If $\Psi, x:\tau \vdash e : \tau'$ and $\sigma <: \Psi, L:\tau \# S \cup L'$ then there exists σ', Θ such that $\sigma, x \in L, e \Downarrow^\star \sigma', L', \Theta$ and $\sigma' <: \Psi, L':\tau' \# S$*

Proof. For part (1), proof is by induction on the structure of derivations of $\Psi \vdash e : \tau$.

- If the expression is a term t then we have

$$\frac{\Psi \vdash_{\mathsf{term}} t : \tau}{\Psi \vdash t : \tau}$$

Hence, $\Psi \vdash_{\mathsf{con}} \mathrm{op}(\sigma, t) : \tau$ so

$$\frac{}{\sigma, l \Leftarrow t \Downarrow \sigma[l := \mathrm{op}(\sigma, t)], l \leftarrow t}$$

where $\sigma[l := \mathrm{op}(\sigma, t)] <: \Psi, l:\tau \# S$.

- If the derivation is of the form

$$\frac{\Psi \vdash l' : \tau_1 \times \tau_2}{\Psi \vdash \pi_i(l') : \tau_i}$$

then we know $\Psi \vdash_{\mathsf{con}} \sigma(l') : \tau_1 \times \tau_2$ so we must have $\sigma(l') = (l_1, l_2)$. Hence, we can derive

$$\frac{\sigma(l') = (l_1, l_2)}{\sigma, l \Leftarrow \pi_i(l') \Downarrow \sigma[l := \sigma(l_i)], l \leftarrow \mathsf{proj}_i(l', l_i)}$$

where $\sigma[l := \sigma(l_i)] <: \Psi, l:\tau_i \# S$.

- If the derivation is of the form

$$\frac{\Psi \vdash e_1 : \tau' \quad \Psi, x:\tau' \vdash e_2 : \tau}{\Psi \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}$$

then choose a fresh $l' \notin \mathrm{dom}(\sigma) \cup S \cup \{l\}$. By induction we have $\sigma, l' \Leftarrow e_1 \Downarrow \sigma', T_1$ where $\sigma' <: \Psi, l':\tau' \# S \cup \{l\}$. Substituting l' for x, we have $\Psi, l':\tau' \vdash e_2[l/x] : \tau$ so by induction we also have $\sigma', l \Leftarrow e_2[l'/x] \Downarrow \sigma'', T_2$ where $\sigma'' <: \Psi, l':\tau', l:\tau \# S$. Finally we can derive

$$\frac{l'\ \mathsf{fresh} \quad \sigma, l' \Leftarrow e_1 \Downarrow \sigma', T_1 \quad \sigma', l \Leftarrow e_2[l'/x] \Downarrow \sigma'', T_2}{\sigma, l \Leftarrow \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow \sigma'', T_1; T_2}$$

and $\sigma <: \Psi, l:\tau \# S$.

- If the derivation is of the form

$$\frac{\Psi(l') = \mathsf{bool} \quad \Psi \vdash e_{\mathsf{t}} : \tau \quad \Psi \vdash e_{\mathsf{f}} : \tau}{\Psi \vdash \mathsf{if}\ l'\ \mathsf{then}\ e_{\mathsf{t}}\ \mathsf{else}\ e_{\mathsf{f}} : \tau}$$

then we must have $\sigma(l') = b \in \mathbb{B}$. By induction, we obtain $\sigma, l \Leftarrow e_b \Downarrow \sigma', T$ where $\sigma' <: \Psi, l:\tau \# S$. Thus, we can conclude

$$\frac{\sigma(l) = b \quad \sigma, l \Leftarrow e_b \Downarrow \sigma', T}{\sigma, l \Leftarrow \mathsf{if}\ l'\ \mathsf{then}\ e_{\mathsf{t}}\ \mathsf{else}\ e_{\mathsf{f}} \Downarrow \sigma', \mathsf{cond}_l(l', b, T)_{e_{\mathsf{t}}}^{e_{\mathsf{f}}}}$$

- If the derivation is of the form

$$\frac{\Psi(l) = \{\tau'\} \quad \Psi, x:\tau' \vdash e : \{\tau\}}{\Psi \vdash \bigcup \{e \mid x \in l\} : \{\tau\}}$$

then we must have $\sigma(l) = L$ where $\Psi \vdash_{\mathsf{con}} L' : \{\tau'\}$. Then there exist σ', L', Θ such that $\sigma, x \in \sigma(l), e \Downarrow^\star \sigma', L', \Theta$ and $\sigma <: \Psi, L':\{\tau'\} \# \{l'\} \cup S$. Hence we can conclude

$$\frac{\sigma, x \in \sigma(l), e \Downarrow^\star \sigma', L', \Theta}{\sigma, l' \Leftarrow \bigcup \{e \mid x \in l\} \Downarrow \sigma'[l' := \bigsqcup \sigma'[L']], l \leftarrow \mathsf{comp}(l, \Theta)_{x.e}}$$

and $\sigma <: \Psi, l':\{\tau'\} \# S$.

- The case for $\sum \{e \mid x \in l\}$ is similar.

For part (2), the proof is by induction on L:

- If $L = \emptyset$ then we can immediately conclude

$$\frac{}{\sigma, x \in \emptyset, e \Downarrow^\star \sigma, \emptyset, \emptyset}$$

where $\sigma <: \Psi \# S$.
- If $L = L_1 \oplus L_2$ then by induction we have $\sigma, x \in L_1, e \Downarrow^\star \sigma_1, L'_1, \Theta_1$ where $\sigma_1 <: \Psi, L_1:\tau' \# S$. Moreover, we also have $\sigma, x \in L_2, e \Downarrow^\star \sigma_2, L'_2, \Theta_2$ where $\sigma_2 <: \Psi, L_2:\tau' \# (\mathrm{dom}(\sigma_1) - \mathrm{dom}(\sigma)) \cup S$. Thus, $\sigma_1 \uplus_\sigma \sigma_2$ exists and avoids S; hence,

$$\frac{\sigma, x \in L_1, e \Downarrow^\star \sigma_1, L'_1, \Theta_1 \quad \sigma, x \in L_2, e \Downarrow^\star \sigma_2, L'_2, \Theta_2}{\sigma, x \in L_1 \oplus L_2, e \Downarrow^\star \sigma_1 \uplus_\sigma \sigma_2, L'_1 \oplus L'_2, \Theta_1 \oplus \Theta_2}$$

and $\sigma_1 \uplus_\sigma \sigma_2 <: \Psi, L_1 \cup L_2:\tau' \# S$.
- If $L = \{l : m\}$ then we can substitute to obtain $\Psi, l:\tau \vdash e[l/x] : \tau'$. Choose l' fresh for $\mathrm{dom}(\sigma) \cup S$ so that we have $\sigma <: \Psi, l:\tau \# S \cup \{l'\}$. Then by induction we have $\sigma, l' \Leftarrow e[l/x] \Downarrow \sigma', T$ where $\sigma' <: \Psi, l:\tau, l':\tau' \# S$. Then we can conclude

$$\frac{l'\ \mathsf{fresh} \quad \sigma, l' \Leftarrow e[l/x] \Downarrow \sigma', T}{\sigma, x \in \{l : m\}, e \Downarrow^\star \sigma', \{l' : m\}, \{[l]T : m\}}$$

since $\sigma' <: \Psi, l':\tau' \# S$.

$\square$

Theorem 6 (Soundness of traced evaluation). *Let* Ψ, e, τ, l, σ *be arbitrary.*

1. *If* $\Psi \vdash e : \tau$ *and* $\sigma, l \Leftarrow e \Downarrow \sigma', T$ *and* $\sigma <: \Psi$ *then* $\Psi \vdash T \triangleright l : \tau$ *and* $\sigma' <: \Psi, l{:}\tau$.
2. *If* $\Psi, x{:}\tau \vdash e : \tau'$ *and* $\sigma <: \Psi, L : \tau$ *and* $\sigma, x{\in}L, e \Downarrow^\star \sigma', L', \Theta$ *then* $\Psi \vdash \tau \triangleright \Theta \triangleright \tau'$ *and* $\sigma' <: \Psi, L' : \tau'$.

Proof. For part (1), proof is by induction on the second derivation.

- If the derivation is of the form

$$\overline{\sigma, l \Leftarrow t \Downarrow \sigma[l := \mathrm{op}(t, \sigma)], l \leftarrow t}$$

then by inversion we have that $\Psi \vdash_{\mathsf{term}} t : \tau$ and so we can derive

$$\frac{\Psi \vdash_{\mathsf{term}} t : \tau}{\Psi \vdash l \leftarrow t \triangleright l : \tau}$$

- If the derivation is of the form

$$\frac{\sigma(l') = (l_1, l_2)}{\sigma, l \Leftarrow \pi_i l' \Downarrow \sigma[l := \sigma(l_i)], l \leftarrow \mathsf{proj}_i(l', l_i)}$$

then by inversion we have that $\Psi(l') = \tau_1 \times \tau_2$, so we may conclude:

$$\frac{\Psi(l') = \tau_1 \times \tau_2}{\Psi \vdash l \leftarrow \mathsf{proj}_i(l', l_i) \triangleright l : \tau_i}$$

- If the derivation is of the form

$$\frac{\sigma, l' \Leftarrow e_1 \Downarrow \sigma_1, T_1 \quad \sigma, l \Leftarrow e_2[l'/x] \Downarrow \sigma_2, T_2}{\sigma, l \Leftarrow \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow \sigma_2, T_1; T_2}\ l'\ \text{fresh}$$

then we must also have

$$\frac{\Psi \vdash e_1 : \tau' \quad \Psi, x{:}\tau' \vdash e_2 : \tau}{\Psi \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}$$

and by induction and substituting l' for x we have $\Psi \vdash T_1 \triangleright l' : \tau'$ and $\Psi, l'{:}\tau' \vdash T_2 \triangleright l : \tau$. So we may conclude

$$\frac{\Psi \vdash T_1 \triangleright l' : \tau' \quad \Psi, l'{:}\tau' \vdash T_2 \triangleright l : \tau}{\Psi \vdash T_1; T_2 \triangleright l : \tau}$$

- If the derivation is of the form:

$$\frac{\sigma(l') = b \quad \sigma, l \Leftarrow e_b \Downarrow \sigma', T}{\sigma, l \Leftarrow \mathsf{if}\ l'\ \mathsf{then}\ e_\mathsf{t}\ \mathsf{else}\ e_\mathsf{f} \Downarrow \sigma', \mathsf{cond}_l(l', b, T)^{e_\mathsf{f}}_{e_\mathsf{t}}}$$

then by inversion we must have

$$\frac{\Psi(l') = \mathsf{bool} \quad \Psi \vdash e_\mathsf{t} : \tau \quad \Psi \vdash e_\mathsf{f} : \tau}{\Psi \vdash \mathsf{if}\ l'\ \mathsf{then}\ e_\mathsf{t}\ \mathsf{else}\ e_\mathsf{f} : \tau}$$

Hence whatever the value of b, by induction we can obtain $\Psi \vdash T \triangleright l : \tau$. To conclude, we derive:

$$\frac{\Psi(l') = \mathsf{bool} \quad \Psi \vdash T \triangleright l : \tau \quad \Psi \vdash e_\mathsf{t} : \tau \quad \Psi \vdash e_\mathsf{f} : \tau}{\Psi \vdash \mathsf{cond}_l(l', b, T)^{e_\mathsf{f}}_{e_\mathsf{t}} \triangleright l : \tau}$$

- If the derivation is of the form

$$\frac{\sigma, x{\in}\sigma(l'), e \Downarrow^\star \sigma', L', \Theta}{\sigma, l \Leftarrow \bigcup\{e \mid x \in l'\} \Downarrow \sigma'[l := \bigsqcup \sigma'[L']], l \leftarrow \mathsf{comp}(l', \Theta)_{x.e}}$$

then by inversion we have

$$\frac{\Psi(l') = \{\tau'\} \quad \Psi, x{:}\tau' \vdash e : \{\tau\}}{\Psi \vdash \bigcup\{e \mid x \in l'\} : \{\tau\}}$$

Then by induction hypothesis (2) we have that $\Psi \vdash \tau' \triangleright \Theta \triangleright \{\tau\}$, so we may conclude:

$$\frac{\Psi(l') = \{\tau'\} \quad \Psi \vdash \tau' \triangleright \Theta \triangleright \{\tau\} \quad \Psi, x{:}\tau' \vdash e : \{\tau\}}{\Psi \vdash l \leftarrow \mathsf{comp}(l', \Theta)_{x.e} \triangleright l : \{\tau\}}$$

- For the $\sum$ case,

$$\frac{\sigma, x{\in}\sigma(l'), e \Downarrow^\star \sigma', L', \Theta}{\sigma, l \Leftarrow \sum\{e \mid x \in l'\} \Downarrow \sigma'[l := \sum \sigma'[L']], l \leftarrow \mathsf{sum}(l', \Theta)_{x.e}}$$

the reasoning is similar to the previous case.

For part (2), proof is by induction on the structure of the third derivation.

- If the derivation is of the form:

$$\overline{\sigma, x{\in}\emptyset, e \Downarrow^\star \sigma, \emptyset, \emptyset}$$

then we can immediately derive

$$\overline{\Psi \vdash \tau \triangleright \emptyset \triangleright \tau'}$$

- If the derivation is of the form:

$$\frac{\sigma, l' \Leftarrow e[l/x] \Downarrow \sigma', T}{\sigma, x{\in}\{l : m\}, e \Downarrow^\star \sigma', \{l' : m\}, \{[l]T : m\}}$$

then we may substitute l for x to obtain $\Psi, l{:}\tau \vdash e[l/x] : \tau'$ and so by induction hypothesis (1) we have $\Psi, l{:}\tau \vdash T \triangleright l' : \tau'$. We may conclude by deriving:

$$\frac{\Psi, l{:}\tau \vdash T \triangleright l' : \tau'}{\Psi \vdash \tau \triangleright \{[l]T : m\} \triangleright \tau'}$$

- If the derivation is of the form:

$$\frac{\sigma, x{\in}L_1, e \Downarrow^\star \sigma_1, L'_1, \Theta_1 \quad \sigma, x{\in}L_2, e \Downarrow^\star \sigma_2, L'_2, \Theta_2}{\sigma, x{\in}L_1 \oplus L_2, e \Downarrow^\star \sigma_1 \uplus_\sigma \sigma_2, L'_1 \oplus L'_2, \Theta_1 \oplus \Theta_2}$$

then by induction we obtain $\Psi \vdash \tau \triangleright \Theta_1 \triangleright \tau'$ and $\Psi \vdash \tau \triangleright \Theta_2 \triangleright \tau'$ so conclude

$$\frac{\Psi \vdash \tau \triangleright \Theta_1 \triangleright \tau' \quad \Psi \vdash \tau \triangleright \Theta_2 \triangleright \tau'}{\Psi \vdash \tau \triangleright \Theta_1 \oplus \Theta_2 \triangleright \tau'}$$

$\square$

We define the set of labels *written* by T, or $\mathrm{Wr}(T)$, as follows:

$$
\begin{aligned}
\mathrm{Wr}(l \leftarrow t) &= \{l\} \\
\mathrm{Wr}(l \leftarrow \mathsf{proj}_i(l', l_i)) &= \{l\} \\
\mathrm{Wr}(\mathsf{cond}_l(l', b, T)^{e_2}_{e_1}) &= \{l\} \cup \mathrm{Wr}(T) \\
\mathrm{Wr}(T_1; T_2) &= \mathrm{Wr}(T_1) \cup \mathrm{Wr}(T_2) \\
\mathrm{Wr}(l \leftarrow \mathsf{comp}(l', \Theta)_{x.e}) &= \{l\} \cup \mathrm{Wr}(\Theta) \\
\mathrm{Wr}(l \leftarrow \mathsf{sum}(l', \Theta)_{x.e}) &= \{l\} \cup \mathrm{Wr}(\Theta) \\
\mathrm{Wr}(\Theta) &= \bigcup\{\mathrm{Wr}(T) \mid [l]T : m \in \Theta\}
\end{aligned}
$$

Finally, we show that the adaptive semantics always succeeds for well-formed traces T and well-formed stores that avoid the labels written by T.

Theorem 7 (Adaptability). *Let S be a finite set of labels, and Ψ, T, τ, l, σ be arbitrary.*

1. *If* $\Psi \vdash T \triangleright l : \tau$ *and* $\sigma <: \Psi \# S \cup \mathrm{Wr}(T)$ *then there exists* σ', T' *such that* $\sigma, T \curvearrowright \sigma', T'$ *and* $\sigma' <: \Psi, l{:}\tau \# S$.
2. *If* $\Psi \vdash \tau \triangleright \Theta \triangleright \tau'$ *and* $\Psi, x{:}\tau \vdash e : \tau'$ *and* $\sigma <: \Psi, L : \tau \# \mathrm{Wr}(\Theta) \cup S$ *then there exist* σ', L', Θ' *such that* $\sigma, x{\in}L, e, \Theta \curvearrowright^\star \sigma', L', \Theta'$ *and* $\sigma' <: \Psi, L'{:}\tau' \# S$.

Proof. For the first part, proof is by induction on the structure of the first derivation.

- If the derivation is of the form

$$\frac{\Psi \vdash_{\mathsf{term}} t : \tau}{\Psi \vdash l \leftarrow t \triangleright l : \tau}$$

  then we can conclude

$$\overline{\sigma, l \leftarrow t \curvearrowright \sigma[l := \mathrm{op}(t, \sigma)], l \leftarrow t}$$

  since σ avoids $\mathrm{Wr}(l \leftarrow t) = \{l\}$. Moreover, $\sigma <: \Psi, l{:}\tau \# S$.

- If the derivation is of the form

$$\frac{\Psi(l') = \tau_1 \times \tau_2}{\Psi \vdash l \leftarrow \mathsf{proj}_i(l', l_i) \triangleright l : \tau_i}$$

  then $\sigma(l')$ must be a pair (l'_1, l'_2), and we can conclude

$$\frac{\sigma(l') = (l'_1, l'_2)}{\sigma, l \leftarrow \mathsf{proj}_i(l', l_i) \curvearrowright \sigma[l := \sigma(l'_i)], l \leftarrow \mathsf{proj}_i(l', l'_i)}$$

  since σ avoids $\mathrm{Wr}(l \leftarrow \mathsf{proj}_i(l', l_i)) = \{l\}$. Note that we do not re-use l_i so the typing judgment does not need to check that it is of the right type. In fact, l_i need not be in Ψ at all. Finally, $\sigma' <: \Psi, l{:}\tau_i \# S$.

- If the derivation is of the form

$$\frac{\Psi \vdash T_1 \triangleright l' : \tau' \quad \Psi, l'{:}\tau' \vdash T_2 \triangleright l : \tau}{\Psi \vdash T_1; T_2 \triangleright l : \tau}$$

  then since $l' \in \mathrm{Wr}(T_1)$ and $\sigma <: \Psi \# \mathrm{Wr}(T_1) \cup (\mathrm{Wr}(T_2) \cup S)$, by induction we have that $\sigma, T_1 \curvearrowright \sigma', T'_1$ and $\sigma' <: \Psi, l'{:}\tau' \# \mathrm{Wr}(T_2) \cup S$. Moreover, since $\sigma' <: \Psi, l'{:}\tau' \# \mathrm{Wr}(T_2) \cup S$ by induction we have $\sigma', T_2 \curvearrowright \sigma'', T'_2$ and $\sigma'' <: \Psi, l'{:}\tau', l{:}\tau \# S$. Hence we may derive

$$\frac{\sigma, T_1 \curvearrowright \sigma', T'_1 \quad \sigma', T_2 \curvearrowright \sigma'', T'_2}{\sigma, T_1; T_2 \curvearrowright \sigma'', T'_1; T'_2}$$

  and also we have $\sigma'' <: \Psi, l{:}\tau \# S$ as desired.

- If the derivation is of the form

$$\frac{\Psi(l') = \mathsf{bool} \quad \Psi \vdash T \triangleright l : \tau \quad \Psi \vdash e_{\mathsf{t}} : \tau \quad \Psi \vdash e_{\mathsf{f}} : \tau}{\Psi \vdash \mathsf{cond}_l(l', b, T)_{e_{\mathsf{t}}}^{e_{\mathsf{f}}} \triangleright l : \tau}$$

  then we must have $\sigma(l') \in \mathbb{B}$. There are two cases. Suppose $\sigma(l) = b$. Then by induction we have that $\sigma, T \curvearrowright \sigma', T'$ and $\sigma' <: \Psi, l{:}\tau \# S$. We can conclude

$$\frac{\sigma(l') = b \quad \sigma, T \curvearrowright \sigma', T'}{\sigma, \mathsf{cond}_l(l', b, T)_{e_{\mathsf{t}}}^{e_{\mathsf{f}}} \curvearrowright \sigma', \mathsf{cond}_l(l', b, T')_{e_{\mathsf{t}}}^{e_{\mathsf{f}}}}$$

  Otherwise, $\sigma(l') = b' \neq b$. So using Theorem 5, we have σ', T' such that $\sigma, l \Leftarrow e_{b'} \Downarrow \sigma', T'$ and $\sigma' <: \Psi, l{:}\tau \# S$, so we may conclude

$$\frac{\sigma(l') = b' \neq b \quad \sigma, l \Leftarrow e_{b'} \Downarrow \sigma', T'}{\sigma, \mathsf{cond}_l(l', b, T)_{e_{\mathsf{t}}}^{e_{\mathsf{f}}} \curvearrowright \sigma', \mathsf{cond}_l(l', b, T')_{e_{\mathsf{t}}}^{e_{\mathsf{f}}}}$$

- If the derivation is of the form

$$\frac{\Psi(l') = \{\tau'\} \quad \Psi \vdash \tau' \triangleright \Theta \triangleright \{\tau\} \quad \Psi, x{:}\tau' \vdash e : \{\tau\}}{\Psi \vdash l \leftarrow \mathsf{comp}(l', \Theta)_{x.e} \triangleright l : \{\tau\}}$$

  then for $L = \sigma(l')$, since $\Psi \vdash_{\mathsf{con}} \sigma(l') : \{\tau'\}$ we have $\sigma <: \Psi, L : \tau' \# \mathrm{Wr}(\Theta) \cup S$. Hence by induction we have σ', L', Θ' such that $\sigma, x{\in}\sigma(l'), e, \Theta \curvearrowright^\star \sigma', L', \Theta'$ and $\sigma' <: \Psi, L' : \{\tau\} \# S$. Therefore, $\bigsqcup \sigma'[L']$ is well-defined so we can conclude

$$\frac{\sigma, x{\in}\sigma(l'), e, \Theta \curvearrowright^\star \sigma', L', \Theta'}{\sigma, l \leftarrow \mathsf{comp}(l', \Theta)_{x.e} \curvearrowright \sigma'[l := \bigsqcup \sigma'[L']], l \leftarrow \mathsf{comp}(l', \Theta')_{x.e}}$$

- If the derivation is of the form

$$\frac{\Psi(l') = \{\tau'\} \quad \Psi \vdash \tau' \triangleright \Theta \triangleright \mathsf{int} \quad \Psi, x{:}\tau' \vdash e : \mathsf{int}}{\Psi \vdash l \leftarrow \mathsf{sum}(l', \Theta)_{x.e} \triangleright l : \mathsf{int}}$$

  then the reasoning is similar to the previous case.

For part (2), the proof is by induction on the structure of L.

- If $L = \emptyset$, then then we can simply conclude

$$\overline{\sigma, x{\in}\emptyset, e, \Theta \curvearrowright^\star \emptyset, \emptyset,}$$

- If $L = \{l : m\}$ then there are two cases. If $[l]T \in \Theta$ for some T, then we proceed as follows. Let $l' = \mathrm{out}(T)$. By Lemma 3, we have that $\Psi, l{:}\tau \vdash e[l/x] \triangleright l' : \tau'$. So, by induction hypothesis (1), we have $\sigma, T \curvearrowright \sigma', T'$ where $\sigma' <: \Psi, l'{:}\tau' \# S$. To conclude, we derive:

$$\frac{[l]T \in \Theta \quad \sigma, T \curvearrowright \sigma', T'}{\sigma, x{\in}\{l : m\}, e, \Theta \curvearrowright^\star \sigma', \{l' : m\}, \{[l]T' : m\}}$$

  Otherwise, $l \notin \mathrm{in}^\star(\Theta)$, so we fall back on traced evaluation. Choose l' fresh for l, σ and S. Since $\sigma <: \Psi, l{:}\tau \# S$, by Theorem 5 we can obtain $\sigma, l' \Leftarrow e \Downarrow \sigma', T'$ where $\sigma' <: \Psi, l'{:}\tau' \# S$. To conclude we derive

$$\frac{l \notin \mathrm{in}^\star(\Theta) \quad l' \text{ fresh} \quad \sigma, l' \Leftarrow e[l/x] \Downarrow \sigma', T'}{\sigma, x{\in}\{l : m\}, e, \Theta \curvearrowright^\star \sigma', \{l' : m\}, \{[l]T' : m\}}$$

- If $L = L_1 \oplus L_2$, then clearly, $\sigma <: \Psi, L_1 : \tau \# \mathrm{Wr}(Ts) \cup S$ so by induction we have $\sigma, x{\in}L_1, e, \Theta \curvearrowright^\star \sigma_1, L'_1, \Theta_1$ where $\sigma_1 <: \Psi, L'_1 : \tau' \# S$. Similarly, we have $\sigma, x{\in}L_2, e, \Theta \curvearrowright^\star \sigma_2, L'_2, \Theta_2$ where $\sigma_2 <: \Psi, L'_2 : \tau' \# (\mathrm{dom}(\sigma_1) - \mathrm{dom}(\sigma)) \cup S$. Hence, σ_1 and σ_2 are orthogonal extensions of σ, so $\sigma_1 \uplus_\sigma \sigma_2$ exists and $\sigma_1 \uplus_\sigma \sigma_2 <: \Psi, L'_1 \cup L'_2 : \tau' \# S$. We conclude by deriving:

$$\frac{\sigma, x{\in}L_1, e, \Theta \curvearrowright^\star \sigma_1, L'_1, \Theta_1 \quad \sigma, x{\in}L_2, e, \Theta \curvearrowright^\star \sigma_2, L'_2, \Theta_2}{\sigma, x{\in}L_1 \oplus L_2, e, \Theta \curvearrowright^\star \sigma_1 \uplus_\sigma \sigma_2, L'_1 \oplus L'_2, \Theta_1 \oplus \Theta_2}$$

$\square$

By combining the above partial fidelity and soundness theorems, we can finally obtain our main result:

Corollary 1 (Total Fidelity). *Suppose* $\sigma_1, l \Leftarrow e \Downarrow \sigma'_1, T_1$ *where* $\sigma_1 : \Psi$ *and* $\Psi \vdash e : \tau$ *and suppose* $\sigma_2 <: \Psi \# \mathrm{Wr}(T)$. *Then there exists* σ'_2, T_2 *such that* $\sigma_2, T_1 \curvearrowright \sigma'_2, T_2$ *and* $\sigma_2, l \Leftarrow e \Downarrow \sigma'_2, T_2$.

Proof. By Theorem 6 we have that $\Psi \vdash T_1 \triangleright l : \tau$. Thus, by Theorem 7 there must exist T_2, σ'_2 such that $\sigma_2, T_1 \curvearrowright \sigma'_2, T_2$. By Theorem 4, it follows that $\sigma_2, l \Leftarrow e \Downarrow \sigma'_2, T_2$. $\square$

5. Provenance extraction

As we discussed in Section 1, a number of forms of provenance have been defined already in the literature. Although most of this work has focused on flat relational queries, several techniques have recently been extended to the NRC. Thus, a natural question is: are traces related to these other forms of provenance?

In this section we describe algorithms for extracting where-provenance (Buneman et al. 2007), dependency provenance (Cheney et al. 2007), and semiring provenance (Foster et al. 2008) from traces. We will develop extraction algorithms and prove them correct relative to the existing definitions. However, our operational formulation of traces is rather different from existing denotational presentations of provenance semantics, so we need to set up appropriate correspondences between store-based and value-based representations. Precisely formulating these equivalences requires introducing several auxiliary definitions and properties.

We also discuss how provenance extraction yields insight into the meaning of other forms of provenance. We can view the extraction algorithms as dynamic analyses of the provenance trace. For example, where-provenance can be viewed an analysis that identifies "chains of copies" form the input to the output. Conversely, we can view high-level properties of traces as clear specifications that can be used to justify new provenance-tracking techniques.

We first fix some terminology used in the rest of the section. We consider an *annotated store* $\sigma^{(h)}$ to consist of a store σ and a function $h : \operatorname{dom}(\sigma) \to A$ mapping each label in σ to an annotation in A. We also consider several kinds of *annotated values*. In general, a value $v \in Val^{(A)}$ with annotations a from some set A is an expression of the form

$$
\begin{aligned}
v &::= w^x \\
w &::= i \mid b \mid (v_1, v_2) \mid \{v_1, \ldots, v_n\}
\end{aligned}
$$

This syntax strictly generalizes that of ordinary values since ordinary values can be viewed as values annotated by elements of some unit set $\{\star\}$, up to an obvious isomorphism. Also, we write $|v|$ for the ordinary value obtained by erasing the annotations from v. This is defined as:

$$
\begin{aligned}
|i^x| &= i \quad |b^x| = b \quad |(v_1, v_2)^x| = (|v_1|, |v_2|) \\
|\{v_1, \ldots, v_n\}| &= \{|v_1|, \ldots, |v_n|\}
\end{aligned}
$$

Moreover, we define $\lfloor w^x \rfloor = w$ and $\lceil w^x \rceil = x$.

Given an A-annotated store $\sigma^{(h)}$, we can extract annotated values using the same technique as extracting ordinary values from an ordinary store:

$$
\begin{aligned}
\sigma^{(h)} \uparrow_{\mathsf{int}}^A l &= \sigma(l)^{h(l)} \\
\sigma^{(h)} \uparrow_{\mathsf{bool}}^A l &= \sigma(l)^{h(l)} \\
\sigma^{(h)} \uparrow_{\tau_1 \times \tau_2}^A l &= (\sigma^{(h)} \uparrow_{\tau_1}^A l_1, \sigma^{(h)} \uparrow_{\tau_2}^A l_2)^{h(l)} \quad (\sigma(l) = (l_1, l_2)) \\
\sigma^{(h)} \uparrow_{\{\tau\}}^A l &= \{\sigma^{(h)} : m \uparrow_\tau^A l' \mid l' : m \in \sigma(l)\}^{h(l)}
\end{aligned}
$$

Moreover, for $\gamma : Var \to Lab$ we again write $\sigma^{(h)} \uparrow_\Gamma^A \gamma : Var \to Val^{(A)}$ for the extension of the annotated value extraction function from labels to environments. Similarly, for L a collection of labels we write $\sigma^{(h)} \uparrow_{\{\tau\}}^A L$ for $\{\sigma \uparrow_\tau^A l : m \mid l : m \in L\}$.

5.1 Where-provenance

As discussed by (Buneman et al. 2001, 2007), where-provenance is information about "where an output value came from in the input". Buneman et al. (2007) defined where-provenance semantics for NRC queries via values annotated with optional annotations $A_\perp = A \uplus \{\perp\}$. Here, $\perp$ stands for the absence of where-provenance, and A is a set of tokens chosen to uniquely address each part of the input.

The idea of where-provenance is that values "copied" via variable or projection expressions retain their annotations, while other operations produce results annotated with $\perp$. We use an auxiliary function

$$
\begin{aligned}
\mathrm{where}(l, h) &= h(l) \\
\mathrm{where}(t, h) &= \perp \quad (t \neq l)
\end{aligned}
$$

that defines the annotation of the result of a term t with respect to $h : Lab \to A_\perp$ to be preserved if $t = l$ and otherwise $\perp$. Buneman et al. (2007) did not consider integer operations or sums; we support them by annotating the results with $\perp$.

We first review the denotational presentation of where-provenance from (Buneman et al. 2007). Figure 14 shows the semantics of expressions e as a function $W[\![e]\!]$ mapping contexts $\gamma : Var \to Val^{(A_\perp)}$ to $A_\perp$-annotated values.

$$
\frac{}{\sigma^{(h)}, l \leftarrow t \Downarrow_W \sigma[l := t]^{(h[l := \mathrm{where}(t,h)])}}
$$

$$
\frac{\sigma^{(h)}, l' \leftarrow e_1 \Downarrow_W \sigma'^{(h')} \quad \sigma'^{(h')}, l \leftarrow e_2[l'/x] \Downarrow_W \sigma''^{(h'')} \quad l' \text{ fresh}}{\sigma^{(h)}, l \leftarrow \mathsf{let}\ x = e_1 \mathsf{\ in\ } e_2 \Downarrow_W \sigma''^{(h'')}}
$$

$$
\frac{\sigma(l') = (l_1, l_2)}{\sigma^{(h)}, l \leftarrow \pi_i(l') \Downarrow_W \sigma[l := \sigma(l_i)]^{(h[l := h(l_i)])}}
$$

$$
\frac{\sigma(l') = b \quad \sigma^{(h)}, l \leftarrow e_b \Downarrow_W \sigma'^{(h')}}{\sigma^{(h)}, l \leftarrow \mathsf{if}\ l' \mathsf{\ then\ } e_t \mathsf{\ else\ } e_f \Downarrow_W \sigma'^{(h')}}
$$

$$
\frac{\sigma^{(h)}, x \in \sigma(l), e \Downarrow_W^\star \sigma'^{(h')}, L'}{\sigma^{(h)}, l \leftarrow \bigcup\{e \mid x \in l'\} \Downarrow_W \sigma'[l := \bigsqcup \sigma'[L']]^{(h[l := \perp])}}
$$

$$
\frac{\sigma^{(h)}, x \in \sigma(l), e \Downarrow_W^\star \sigma'^{(h')}, L'}{\sigma^{(h)}, l \leftarrow \sum\{e \mid x \in l'\} \Downarrow_W \sigma'[l := \sum \sigma'[L']]^{(h[l := \perp])}}
$$

$$
\frac{}{\sigma^{(h)}, x \in \emptyset, e \Downarrow_W^\star \sigma^{(h)}, \emptyset}
$$

$$
\frac{\sigma^{(h)}, x \in L_1, e \Downarrow_W^\star \sigma_1^{(h_1)}, L_1' \quad \sigma^{(h)}, x \in L_1, e \Downarrow_W^\star \sigma_2^{(h_2)}, L_2'}{\sigma^{(h)}, x \in L_1 \oplus L_2, e \Downarrow_W^\star \sigma_1 \uplus_\sigma \sigma_2^{(h_1 \uplus_h h_2)}, L_1' \oplus L_2'}
$$

$$
\frac{\sigma^{(h)}, l' \leftarrow e[l/x] \Downarrow_W \sigma'^{(h')} \quad l' \text{ fresh}}{\sigma^{(h)}, x \in \{l : m\}, e \Downarrow_W^\star \sigma'^{(h')}, \{l' : m\}}
$$

Figure 15. Where-provenance, operationally

$$
\frac{}{h, l \leftarrow t \rightsquigarrow_W h[l := \mathrm{where}(t, h)]} \quad \frac{h, T_1 \rightsquigarrow_W h' \quad h', T_2 \rightsquigarrow_W h''}{h, T_1; T_2 \rightsquigarrow_W h''}
$$

$$
\frac{}{h, l \leftarrow \mathsf{proj}_i(l', l'') \rightsquigarrow_W h[l := h(l'')]} \quad \frac{h, T \rightsquigarrow_W h'}{h, \mathsf{cond}_l(l', b, T) \rightsquigarrow_W h'}
$$

$$
\frac{h, \Theta \rightsquigarrow_W^\star h'}{h, l \leftarrow \mathsf{comp}(l', \Theta) \rightsquigarrow_W h'[l := \perp]} \quad \frac{h, \Theta \rightsquigarrow_W^\star h'}{h, l \leftarrow \mathsf{sum}(l', \Theta) \rightsquigarrow_W h'[l := \perp]}
$$

$$
\frac{}{h, \emptyset \rightsquigarrow_W^\star h} \quad \frac{h, \Theta_1 \rightsquigarrow_W^\star h_1 \quad h, \Theta_2 \rightsquigarrow_W^\star h_2}{h, \Theta_1 \oplus \Theta_2 \rightsquigarrow_W^\star h_1 \uplus_h h_2} \quad \frac{h, T \rightsquigarrow_W h'}{h, \{[l]T : m\} \rightsquigarrow_W^\star h'}
$$

Figure 16. Extracting where-provenance

In Figure 15, we introduce an equivalent operational formulation. We define judgments $\sigma^{(h)}, l \leftarrow e \Downarrow_W \sigma'^{(h')}$ for expression evaluation and $\sigma^{(h)}, x \in L, e \Downarrow_W^\star \sigma'^{(h')}, L'$ for iteration, both with where-provenance propagation.

It is straightforward to prove by induction that:

Theorem 8.

1. *Suppose $\Gamma \vdash e : \tau$ and $\Psi \vdash \sigma, \gamma : \Gamma$. Then $\sigma^{(h)}, l \leftarrow \gamma(e) \Downarrow_W \sigma'^{(h')}$ if and only if $W[\![e]\!](\sigma^{(h)} \uparrow_\Gamma^{A_\perp} \gamma) = \sigma'^{(h')} \uparrow_\tau^{A_\perp} l$.*

2. *Suppose $\Gamma, x : \tau \vdash e : \tau'$ and $\Psi \vdash \sigma, \gamma : \Gamma$. Then $\sigma^{(h)}, x \in L, \gamma(e) \Downarrow_W^\star \sigma'^{(h')}, L'$ if and only if $\{W[\![e]\!]\gamma[x := v] \mid v \in \sigma^{(h)} \uparrow_{\{\tau\}}^{A_\perp} L\} = \sigma'^{(h')} \uparrow_{\{\tau'\}}^{A_\perp} L'$.*

The where-provenance extraction relation is shown in Figure 16; we define judgment $h, T \rightsquigarrow_W h'$, which takes input annotations h and propagates them through T to yield output annotations h', and judgment $h, \Theta \rightsquigarrow_W^\star h'$ which propagates annotations through a set of traces. Where-provenance extraction can be shown correct relative to the operational where-provenance semantics, as follows:

Theorem 9. *1. Suppose $\sigma, l \Leftarrow e \Downarrow \sigma', T$ and $h : \operatorname{dom}(\sigma) \to A_\perp$ is given. Then $\sigma^{(h)}, l \leftarrow e \Downarrow_W \sigma'^{(h')}$ holds if and only if $h, T \rightsquigarrow_W h'$ holds.*
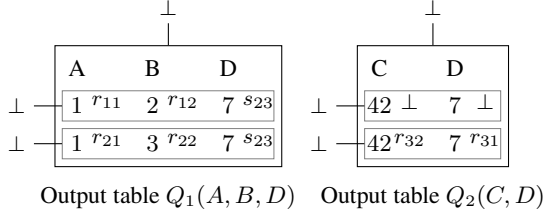
$$\bot \qquad\qquad \bot$$

A	B	D		C	D
$1\ ^{r_{11}}$	$2\ ^{r_{12}}$	$7\ ^{s_{23}}$	$42\ \bot$	$7\ \bot$
$1\ ^{r_{21}}$	$3\ ^{r_{22}}$	$7\ ^{s_{23}}$	$42\ ^{r_{32}}$	$7\ ^{r_{31}}$

Output table $Q_1(A, B, D)$ Output table $Q_2(C, D)$

Figure 17. Where-provenance extraction examples

2. *If $\sigma, x \in L, e \Downarrow^{\star} \sigma', L'$, then $\sigma^{(h)}, x \in L, e \Downarrow^{\star}_W \sigma'^{(h')}, L'$ if and only if $h, \Theta \rightsquigarrow^{\star}_W h'$.*

Example 4 Figure 17 shows the results of where-provenance extraction for Examples 1–2. For the inputs and results in Figure 1, the field values copied from the input have provenance links to their sources, whereas values computed from several values have no where-provenance ($\bot$).

Definition 2 A *copy* with source l' and target l is a trace of either the form $l \leftarrow l'$ or $l \leftarrow \mathsf{proj}_i(l'', l')$. A *chain of copies* from l_0 to l_n is a sequence of trace steps $T_1; \ldots; T_n$ where each step T_i is a copy from l_{i-1} to l_i. We say that a trace T *contains a chain of copies* from l' to l if there is a chain of copies from l' to l all of whose operations are present in T.

Let $\mathrm{id}_\sigma : \mathrm{dom}(\sigma) \to \mathrm{dom}(\sigma)_\bot$ be the (lifted) identity function on σ.

Proposition 4. *Suppose $\sigma, l \Leftarrow e \Downarrow \sigma', T$ and $\mathrm{id}_\sigma, T \rightsquigarrow_W h$. Then for each $l' \in \mathrm{dom}(\sigma')$, $h(l') \neq \bot$ if and only if there is a chain of copies from $h(l')$ to l' in T.*

Moreover, where-provenance can easily be extracted from a trace for a single input or output label rather than for all of the labels simultaneously, simply by traversing the trace. Though this takes time $O(|T|)$ in the worst case, we could do much better if the traces are represented as graphs rather than as syntax trees.

5.2 Dependency provenance

We next consider extracting the *dependency provenance* introduced in our previous work (Cheney et al. 2007). Dependency provenance is motivated by the concepts of dependency that underlie program slicing (Venkatesh 1991) and noninterference in information flow security, as formalized, for instance, in the Dependency Core Calculus (Abadi et al. 1999). We consider NRC values annotated with sets of tokens and define an annotation-propagating semantics.

Dependency provenance annotations are viewed as correct when they link each part of the input to all parts of the output that *may* change if the input part is changed. This is similar to non-interference. The resulting links can be used to "slice" the input with respect to the output and vice versa. Cheney et al. (2007) established that, as with minimal program slices, minimal dependency provenance is not computable, but gave dynamic and static approximations. Here, we will show how to extract the dynamic approximation from traces.

Dependency provenance can be modeled using values $v \in Val^{(\mathcal{P}(A))}$ annotated with sets of tokens from A. We introduce an auxiliary function $dep(t, h)$ for calculating the dependences of

$$\overline{\sigma^{(h)}, l \leftarrow t \Downarrow_D \sigma[l := t]^{(h[l := \mathrm{dep}(t,h)])}}$$

$$\frac{\sigma^{(h)}, l' \leftarrow e_1 \Downarrow_D \sigma'^{(h')} \quad \sigma'^{(h')}, l \leftarrow e_2[l'/x] \Downarrow_D \sigma''^{(h'')} \quad l' \text{ fresh}}{\sigma^{(h)}, l \leftarrow \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow_D \sigma''^{(h'')}}$$

$$\frac{\sigma(l') = (l_1, l_2)}{\sigma^{(h)}, l \leftarrow \pi_i(l') \Downarrow_D \sigma[l := \sigma(l_i)]^{(h[l := h(l_i) \cup h(l')])}}$$

$$\frac{\sigma(l') = b \quad \sigma^{(h)}, l \leftarrow e_b \Downarrow_D \sigma'^{(h')}}{\sigma^{(h)}, l \leftarrow \mathsf{if}\ l'\ \mathsf{then}\ e_\mathsf{t}\ \mathsf{else}\ e_\mathsf{f} \Downarrow_D \sigma'^{(h'[l := h'(l) \cup h'(l')])}}$$

$$\frac{\sigma^{(h)}, x \in \sigma(l), e \Downarrow^{\star}_D \sigma'^{(h')}, L'^{(a)}}{\sigma^{(h)}, l \leftarrow \bigcup \{e \mid x \in l'\} \Downarrow_D \sigma'[l := \bigsqcup \sigma'[L']]^{(h'[l := h'(l') \cup a])}}$$

$$\frac{\sigma^{(h)}, x \in \sigma(l), e \Downarrow^{\star}_D \sigma'^{(h')}, L'^{(a)}}{\sigma^{(h)}, l \leftarrow \sum \{e \mid x \in l'\} \Downarrow_D \sigma'[l := \sum \sigma'[L']]^{(h'[l := h'(l') \cup a])}}$$

$$\overline{\sigma^{(h)}, x \in \emptyset, e \Downarrow^{\star}_D \sigma^{(h)}, \emptyset^{(\emptyset)}}$$

$$\frac{\sigma^{(h)}, x \in L_1, e \Downarrow^{\star}_D \sigma_1^{(h_1)}, L_1'^{(a_1)} \quad \sigma^{(h)}, x \in L_1, e \Downarrow^{\star}_D \sigma_2^{(h_2)}, L_2'^{(a_2)}}{\sigma^{(h)}, x \in L_1 \oplus L_2, e \Downarrow^{\star}_D \sigma_1 \uplus_\sigma \sigma_2^{(h_1 \uplus_h h_2)}, (L_1' \oplus L_2')^{(a_1 \cup a_2)}}$$

$$\frac{\sigma^{(h)}, l' \leftarrow e[l/x] \Downarrow_D \sigma'^{(h')} \quad l' \text{ fresh}}{\sigma^{(h)}, x \in \{l : m\}, e \Downarrow^{\star}_D \sigma'^{(h')}, \{l' : m\}^{(h'(l'))}}$$

Figure 19. Dependency-provenance, operationally

basic terms t relative to annotation functions $h : Lab \to \mathcal{P}(A)$.

$$
\begin{aligned}
\mathrm{dep}(i, h) = \mathrm{dep}(b, h) = \mathrm{dep}(\emptyset, h) &= \emptyset \\
\mathrm{dep}(\{l\}, h) = \mathrm{dep}(\neg l, h) = \mathrm{dep}(l, h) &= h(l) \\
\mathrm{dep}(\mathsf{empty}(l), h) &= h(l) \\
\mathrm{dep}(l_1 + l_2, h) = \mathrm{dep}(l_1 \approx l_2, h) &= h(l_1) \cup h(l_2) \\
\mathrm{dep}(l_1 \wedge l_2, h) = \mathrm{dep}((l_1, l_2), h) &= h(l_1) \cup h(l_2) \\
\mathrm{dep}(l_1 \cup l_2, h) &= h(l_1) \cup h(l_2)
\end{aligned}
$$

Essentially, dep simply takes the union of the annotations of all labels mentioned in a term.

Cheney et al. (2007) defined dynamic provenance-tracking denotationally as a function $D[\![e]\!]$ mapping contexts $\gamma : Var \to Val^{(\mathcal{P}(A))}$ to $\mathcal{P}(A)$-annotated values. We present this definition in Figure 18. Note that we use an auxiliary notation v^{+a} to indicate adding an annotation to the toplevel of a $\mathcal{P}(A)$-annotated value. That is, $(w^b)^{+a} = w^{b \cup a}$.

Next we introduce an operational version. We define judgments $\sigma^{(h)}, l \leftarrow e \Downarrow_D \sigma'^{(h')}$ for expression evaluation and $\sigma^{(h)}, x \in L, e \Downarrow^{\star}_D \sigma'^{(h')}, L'^{(a)}$ for comprehension evaluation, both with where-provenance propagation. Note that the iteration rules maintain an annotation set a collecting the top-level annotations of the elements of L'.

It is straightforward to prove by induction that:

Theorem 10.

1. *Suppose $\Gamma \vdash e : \tau$ and $\Psi \vdash \sigma, \gamma : \Gamma$. Then $\sigma^{(h)}, l \leftarrow e \Downarrow_D \sigma'^{(h')}$ if and only if $D[\![e]\!](\sigma^{(h)} \uparrow^{\mathcal{P}(A)}_\Gamma \gamma) = \sigma'^{(h')} \uparrow^{\mathcal{P}(A)}_\tau l$.*

2. *Suppose $\Gamma, x : \tau \vdash e : \tau'$ and $\Psi \vdash \sigma, \gamma : \Gamma$. Then $\sigma^{(h)}, x \in L, e \Downarrow^{\star}_D \sigma'^{(h')}, L'^{(a)}$ if and only if $\{D[\![e]\!]\gamma[x := v] \mid v \in \sigma^{(h)} \uparrow^{\mathcal{P}(A)}_{\{\tau\}} L\} = \sigma'^{(h')} \uparrow^{\mathcal{P}(A)}_{\{\tau'\}} L'$ and $a = \cup\{\sigma(l') \mid l' \in L'\}$.*

We define the dependency-provenance extraction judgments $h, T \rightsquigarrow_D h'$ and $h, \Theta \rightsquigarrow^{\star}_D h'$ in Figure 21. As usual, we have

$$
\begin{aligned}
W[\![x]\!]\gamma &= \gamma(x) \\
W[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] &= W[\![e_2]\!]\gamma[x := W[\![e_1]\!]\gamma] \\
W[\![i]\!]\gamma &= i^{\perp} \\
W[\![e_1 + e_2]\!]\gamma &= (\lfloor W[\![e_1]\!]\gamma \rfloor + \lfloor W[\![e_2]\!]\gamma \rfloor)^{\perp} \\
W[\![\textstyle\sum\{e \mid x \in e_0\}]\!]\gamma &= (\textstyle\sum\{\lfloor W[\![e]\!]\gamma[x \mapsto v] \rfloor \mid v \in \lfloor W[\![e_0]\!]\gamma \rfloor\})^{\perp} \\
W[\![b]\!]\gamma &= b^{\perp} \\
W[\![\neg e]\!]\gamma &= (\neg \lceil W[\![e]\!]\gamma \rceil)^{\perp} \\
W[\![e_1 \wedge e_2]\!]\gamma &= (\lceil W[\![e_1]\!]\gamma \rceil \wedge \lceil W[\![e_2]\!]\gamma \rceil)^{\perp} \\
W[\![(e_1, e_2)]\!]\gamma &= (W[\![e_1]\!]\gamma, W[\![e_2]\!]\gamma)^{\perp} \\
W[\![\pi_i(e)]\!]\gamma &= \pi_i(\lfloor W[\![e]\!]\gamma \rfloor) \\
W[\![\emptyset]\!]\gamma &= \emptyset^{\perp} \\
W[\![\{e\}]\!]\gamma &= \{W[\![e]\!]\gamma\}^{\perp} \\
W[\![e_1 \cup e_2]\!]\gamma &= (\lfloor W[\![e_1]\!]\gamma \rfloor \cup \lfloor W[\![e_2]\!]\gamma \rfloor)^{\perp} \\
W[\![\textstyle\bigcup\{e \mid x \in e_0\}]\!]\gamma &= (\textstyle\bigcup\{\lfloor W[\![e]\!]\gamma[x \mapsto v] \rfloor \mid v \in \lfloor W[\![e_0]\!]\gamma \rfloor\})^{\perp} \\
W[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\gamma &= \begin{cases} W[\![e_1]\!]\gamma & \text{if } \lfloor W[\![e_0]\!]\gamma \rfloor = \mathsf{t} \\ W[\![e_2]\!]\gamma & \text{if } \lfloor W[\![e_0]\!]\gamma \rfloor = \mathsf{f} \end{cases} \\
W[\![e_1 \approx e_2]\!]\gamma &= \begin{cases} \mathsf{t}^{\perp} & \text{if } \lfloor W[\![e_1]\!]\gamma \rfloor = \lfloor W[\![e_2]\!]\gamma \rfloor \\ \mathsf{f}^{\perp} & \text{if } \lfloor W[\![e_1]\!]\gamma \rfloor \neq \lfloor W[\![e_2]\!]\gamma \rfloor \end{cases} \\
W[\![\mathsf{empty}(e)]\!]\gamma &= \begin{cases} \mathsf{t}^{\perp} & \text{if } \lfloor W[\![e]\!]\gamma \rfloor = \emptyset \\ \mathsf{f}^{\perp} & \text{if } \lfloor W[\![e]\!]\gamma \rfloor \neq \emptyset \end{cases}
\end{aligned}
$$

Figure 14. Where-provenance, denotationally

$$
\begin{aligned}
\textstyle\bigsqcup^{D}(\{w_1^{a_1} : m_1 \ldots, w_n^{a_n} : m_n\})^a &= (\textstyle\bigsqcup(\{w_1 : m_1, \ldots, w_n : m_n\}))^{a \cup a_1 \cup \cdots \cup a_n} \\
\textstyle\sum^{D}(\{w_1^{a_1} : m_1 \ldots, w_n^{a_n} : m_n\})^a &= (\textstyle\sum(\{w_1 : m_1, \ldots, w_n : m_n\}))^{a \cup a_1 \cup \cdots \cup a_n}
\end{aligned}
$$

$$
\begin{aligned}
D[\![x]\!]\gamma &= \gamma(x) \\
D[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] &= D[\![e_2]\!]\gamma[x := D[\![e_1]\!]\gamma] \\
D[\![i]\!]\gamma &= i^{\emptyset} \\
D[\![e_1 + e_2]\!]\gamma &= D[\![e_1]\!]\gamma +^{D} D[\![e_2]\!]\gamma & w_1^{a_1} +^{D} w_2^{a_2} &= (w_1 + w_2)^{a_1 \cup a_2} \\
D[\![\textstyle\sum\{e \mid x \in e_0\}]\!]\gamma &= \textstyle\sum^{D}\{D[\![e]\!]\gamma[x \mapsto v] \mid v \in D[\![e_0]\!]\gamma\} \\
D[\![b]\!]\gamma &= b^{\emptyset} \\
D[\![\neg e]\!]\gamma &= \neg^{D} D[\![e]\!]\gamma & \neg^{D}(w^a) &= (\neg w)^a \\
D[\![e_1 \wedge e_2]\!]\gamma &= D[\![e_1]\!]\gamma \wedge^{D} D[\![e_2]\!]\gamma & w_1^{a_1} \wedge^{D} w_2^{a_2} &= (w_1 \wedge w_2)^{a_1 \cup a_2} \\
D[\![(e_1, e_2)]\!]\gamma &= (D[\![e_1]\!]\gamma, D[\![e_2]\!]\gamma)^{\emptyset} \\
D[\![\pi_i(e)]\!]\gamma &= \pi_i(\lfloor D[\![e]\!]\gamma \rfloor)^{+\lceil D[\![e]\!]\gamma \rceil} \\
D[\![\emptyset]\!]\gamma &= \emptyset^{\emptyset} \\
D[\![\{e\}]\!]\gamma &= \{D[\![e]\!]\gamma\}^{\emptyset} \\
D[\![e_1 \cup e_2]\!]\gamma &= D[\![e_1]\!]\gamma \cup^{D} D[\![e_2]\!]\gamma & w_1^{a_1} \cup^{D} w_2^{a_2} &= (w_1 \cup w_2)^{a_1 \cup a_2} \\
D[\![\textstyle\bigcup\{e \mid x \in e_0\}]\!]\gamma &= \textstyle\bigsqcup^{D}\{D[\![e]\!]\gamma[x \mapsto v] \mid v \in D[\![e_0]\!]\gamma\} \\
D[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\gamma &= \begin{cases} D[\![e_1]\!]\gamma^{+\lceil D[\![e_0]\!]\gamma \rceil} & \text{if } [\![e_0]\!]\gamma = \mathsf{t} \\ D[\![e_2]\!]\gamma^{+\lceil D[\![e_0]\!]\gamma \rceil} & \text{if } [\![e_0]\!]\gamma = \mathsf{f} \end{cases} \\
D[\![e_1 \approx e_2]\!]\gamma &= D[\![e_1]\!]\gamma \approx^{D} D[\![e_2]\!]\gamma & w_1^{a_1} \approx^{D} w_2^{a_2} &= (w_1 \approx w_2)^{a_1 \cup a_2} \\
D[\![\mathsf{empty}(e)]\!]\gamma &= \mathsf{empty}^{D}(D[\![e]\!]\gamma) & \mathsf{empty}^{D}(w^a) &= (\mathsf{empty}(w))^a
\end{aligned}
$$

Figure 18. Dependency-provenance, denotationally

$$A_1 = \{r, s, r_1, r_2, r_3, s_1, s_2, s_3, r_{13}, s_{11}, r_{23}, s_{21}, r_{33}, s_{31}\}$$
$$A_2 = \{r, s, r_1, r_2, r_3, s_1, s_2, s_3, r_{12}, r_{22}, r_{32}\}$$
$$A_3 = \{s_{11}, s_{12}, s_{21}, s_{22}, s_{31}\}$$

A_1				A_2	

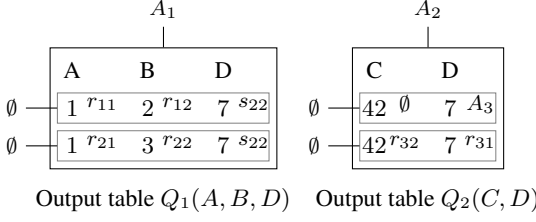|	A	B	D		C	D
∅ —	1 r_{11}	2 r_{12}	7 s_{22}	∅ —	42 ∅	7 A_3
∅ —	1 r_{21}	3 r_{22}	7 s_{22}	∅ —	42 r_{32}	7 r_{31}

Output table $Q_1(A, B, D)$ Output table $Q_2(C, D)$

Figure 20. Dependency provenance extraction examples

$$\frac{}{h, l \leftarrow t \rightsquigarrow_D h[l := \mathsf{dep}(t,h)]} \qquad \frac{h, T_1 \rightsquigarrow_D h' \quad h', T_2 \rightsquigarrow_D h''}{h, T_1; T_2 \rightsquigarrow_D h''}$$

$$\frac{}{h, l \leftarrow \mathsf{proj}_i(l', l_i) \rightsquigarrow_D h[l := h(l') \cup h(l_i)]}$$

$$\frac{h, T \rightsquigarrow_D h'}{h, \mathsf{cond}_l(l', b, T) \rightsquigarrow_D h'[l' := h'(l') \cup h'(l)]}$$

$$\frac{h, \Theta \rightsquigarrow_D^\star h'^{(a)}}{h, l \leftarrow \mathsf{comp}(l, \Theta) \rightsquigarrow_D h'[l := h'(l') \cup a]}$$

$$\frac{h, \Theta \rightsquigarrow_D^\star h'^{(a)}}{h, l \leftarrow \mathsf{sum}(l, \Theta) \rightsquigarrow_D h'[l := h'(l') \cup a]}$$

$$\frac{}{h, \emptyset \rightsquigarrow_D^\star h^{(\emptyset)}} \quad \frac{h, T \rightsquigarrow_D h'}{h, \{[l]T\} \rightsquigarrow_D^\star h'^{(h'(\mathsf{out}(T)))}}$$

$$\frac{h, \Theta_1 \rightsquigarrow_D^\star h_1^{(a_1)} \quad h, \Theta_2 \rightsquigarrow_D^\star h_2^{(a_2)}}{h, \Theta_1 \oplus \Theta_2 \rightsquigarrow_D^\star (h_1 \uplus_h h_2)^{(a_1 \cup a_2)}}$$

Figure 21. Extracting dependency provenance

two judgments, one for traversing traces and another for traversing trace sets.

Theorem 11. *1. Suppose $\sigma, l \Leftarrow e \Downarrow \sigma', T$ and $h : \mathrm{dom}(\sigma) \rightarrow \mathcal{P}(A)$. Then $\sigma^{(h)}, l \leftarrow e \Downarrow_D \sigma'^{(h')}$ holds if and only if $h, T \rightsquigarrow_D h'$ holds.*
2. If $\sigma, x \in L, e \Downarrow^\star \sigma', L', \Theta$ and $h : \mathrm{dom}(\sigma) \rightarrow \mathcal{P}(A)$ then $\sigma^{(h)}, x \in L, e \Downarrow_D^\star \sigma'^{(h')}, L'^{(a)}$ holds if and only if $h, \Theta \rightsquigarrow_D^\star h'^{(a)}$ holds.

Example 5 Figure 20 shows the results of dependency provenance extraction for Examples 1–2. The dependency-provenance is similar to the where-provenance for several fields such as l_{11}. The rows l'_1, l'_2 have no (immediate) dependences. The top-level labels l, l' depend on many parts of the input — essentially on all parts at which changes could lead to global changes to the output table.

5.3 Semiring provenance

Green et al. (2007) introduced the *semiring-annotated relational model*. Recall that a (commutative) semiring is an algebraic structure $(K, 0_K, 1_K, +_K, \cdot_K)$ such that $(K, 0, +)$ and $(K, 1, \cdot)$ are commutative monoids, 0 is an annilhilator (that is, $0 \cdot x = 0 - x \cdot 0$) and $\cdot$ distributes over $+$. They considered K-relations to be ordinary finite relations whose elements are annotated with elements of K, and interpreted relational calculus queries over K-relations such that many known variations of the relational model are a special case. For example, ordinary set-based semantics corresponds to the semiring $(\mathbb{B}, \mathsf{f}, \mathsf{t}, \vee, \wedge)$, whereas the multiset or bag semantics corresponds to the semiring $(\mathbb{N}, 0, 1, +, \cdot)$.

The most general instance of the K-relational model is obtained by taking K to be the *free semiring* $\mathbb{N}[X]$ of polynomials with coefficients in $\mathbb{N}$ over indeterminates X, and Green et al. (2007) considered this to yield a form of provenance that they called *how-provenance* because it provides more information about how a tuple was derived from the input. Lineage and why-provenance can also be obtained as instances of the semiring model (although the initial paper glossed over some subtleties that were later clarified by (Buneman et al. 2008)). Thus, if we can extract semiring provenance from traces, we can also extract lineage and why-provenance.

Foster et al. (2008) extended the semiring-valued model to the NRC, and we will work in terms of this version. Formally, given semiring K, Foster et al. (2008) interpret types as follows:

$$K[\![\mathsf{int}]\!] = \mathbb{Z} \qquad K[\![\mathsf{bool}]\!] = \mathbb{B}$$
$$K[\![\tau_1 \times \tau_2]\!] = K[\![\tau_1]\!] \times K[\![\tau_2]\!]$$
$$K[\![\{\tau\}]\!] = \{f : K[\![\tau]\!] \rightarrow K \mid \mathrm{supp}(f) \text{ finite}\}$$

where $\mathrm{supp}(f) = \{x \in X \mid f(x) \neq 0_K\}$ provided $f : X \rightarrow K$. In other words, integer, boolean and pair types are interpreted normally, and collections of type τ are interpreted as *finitely-supported* functions from $K[\![\tau]\!]$ to K. For example, finitely-supported functions $X \rightarrow \mathbb{B}$ correspond to finite relations over X, whereas finitely-supported functions $X \rightarrow \mathbb{N}$ correspond to finite multisets. We overload the multiset notation $\{v_1 : k_1, \ldots\}$ for K-collections over K-values v to indicate that the annotation of v_i is k_i. We write K-*Val* for the set of all K-values of any type.

We write $\mathcal{K}(X)$ for $\{f : X \rightarrow K \mid \mathrm{supp}(f) \text{ finite}\}$. This forms an additive monad with zero. To simplify notation, we define its "return" ($\eta_\mathcal{K}$), "bind" ($\bullet_\mathcal{K}$), zero ($0_\mathcal{K}$), and addition ($+_\mathcal{K}$) operators as follows:

$$\eta_\mathcal{K}(x) = \lambda y.\text{if } x = y \text{ then } 1_K \text{ else } 0_K$$
$$f \bullet_\mathcal{K} g = \lambda y. \sum_{x \in \mathrm{supp}(f)} f(x) \cdot_K g(x)(y)$$
$$0_\mathcal{K} = \lambda x.0_K$$
$$f +_\mathcal{K} g = \lambda x.f(x) +_K g(x)$$

Moreover, if $f : X \rightarrow K$ and $k \in K$ then we write $k \cdot_\mathcal{K} f$ for the "scalar multiplication" of v by k, that is, $k \cdot f = \lambda x.k \cdot_K f(x)$.

Foster et al. (2008) defined the semantics of NRC over K-values denotationally. Figure 22 presents a simplified version of this semantics in terms of the $\mathcal{K}$ monad operations; we interpret an expression e as a function from environments $\gamma : Var \rightarrow K$-*Val* to results in K-*Val*. Note that Foster et al. (2008)'s version of NRC excludes emptiness tests, integers, booleans and primitive operations other than equality, but also includes some features we do not consider such as a tree type used to model unordered XML. Most of the rules are similar to the ordinary denotational semantics of NRC; only the rules involving collection types are different. A suitable type soundness theorem can be shown easily for this interpretation.

Semiring-valued relations place annotations only on the elements of collections. To model these annotations correctly using stores, we annotate labels of collections with K-collections of labels $\mathcal{K}(Lab)$. As a simple example, consider store $[l_1 := 1, l_2 := 2, l_3 := 1, l := \{l_1 : 2, l_2 : 3, l_3\}]$ and annotation function $h(l) = [l_1 := k_1, l_2 := k_2, l_3 := k_3]$. Then l can be interpreted as the K-value $\{1 : 2k_1 + k_3, 2 : 3k_2\}$. The reason for annotating collections with $\mathcal{K}(Lab)$ instead of annotating collection element labels directly is that due to sharing, a label may be an element of more than one collection in a store (with different K-annotations). For example, consider $[l_1 := 1, l_2 := 2, l := \{l_1 : 2, l_2\}, l' := \{l_1 : 42\}]$. If we annotate l with $[l_1 \mapsto k_1, l_2 \mapsto k_2]$ and l' with $[l_1 := k_3]$ then we can interpret l as $\{1 : 2k_1, 2 : k_2\}$ and l' as $\{1 : 42k_3\}$

$$\begin{aligned}
K[\![x]\!]\gamma &= \gamma(x)\\
K[\![\text{let } x = e_1 \text{ in } e_2]\!]\gamma &= K[\![e_2]\!]\gamma[x \mapsto K[\![e_1]\!]\gamma]\\
K[\![b]\!]\gamma &= b\\
K[\![\neg e]\!]\gamma &= \neg K[\![e]\!]\gamma\\
K[\![e_1 \wedge e_2]\!]\gamma &= K[\![e_1]\!]\gamma \wedge K[\![e_2]\!]\gamma\\
K[\![(e_1, e_2)]\!]\gamma &= (K[\![e_1]\!]\gamma, K[\![e_2]\!]\gamma)\\
K[\![\pi_i(e)]\!]\gamma &= \pi_i(K[\![e]\!]\gamma)\\
K[\![\emptyset]\!]\gamma &= 0_{\mathcal{K}}\\
K[\![\{e\}]\!]\gamma &= \eta_{\mathcal{K}}(K[\![e]\!]\gamma)\\
K[\![e_1 \cup e_2]\!]\gamma &= K[\![e_1]\!]\gamma +_{\mathcal{K}} K[\![e_2]\!]\gamma\\
K[\![\bigcup\{e \mid x \in e_0\}]\!]\gamma &= K[\![e_0]\!]\gamma \bullet_{\mathcal{K}} (\lambda v.K[\![e]\!]\gamma[x \mapsto v])\\
K[\![\text{if } e_0 \text{ then } e_1 \text{ else } e_2]\!]\gamma &= \begin{cases} K[\![e_1]\!]\gamma & \text{if } K[\![e_0]\!]\gamma = \mathsf{t}\\ K[\![e_2]\!]\gamma & \text{if } K[\![e_0]\!]\gamma = \mathsf{f} \end{cases}\\
K[\![e_1 \approx e_2]\!]\gamma &= \begin{cases} \mathsf{t} & \text{if } K[\![e_1]\!]\gamma = K[\![e_2]\!]\gamma\\ \mathsf{f} & \text{if } K[\![e_1]\!]\gamma \neq K[\![e_2]\!]\gamma \end{cases}
\end{aligned}$$

Figure 22. Semiring provenance, denotationally

respectively. If the annotations were placed directly on l_1, l_2 then this would not be possible.

We will consider annotation functions $h : Lab \to \mathcal{K}(Lab)_{\perp}$ such that if l is the label of a collection, then $h(l)$ maps the elements of l to their K-values. Labels of pair, integer, or boolean constructors are mapped to $\perp$. In what follows, we will use an auxiliary function $\mathrm{semiring}(l, h)$ to deal with the basic operations:

$$\begin{aligned}
\mathrm{semiring}(l, h) &= h(l)\\
\mathrm{semiring}(\emptyset, h) &= 0_{\mathcal{K}}\\
\mathrm{semiring}(\{l\}, h) &= \eta_{\mathcal{K}}(l)\\
\mathrm{semiring}(l_1 \cup l_2, h) &= h(l_1) +_{\mathcal{K}} h(l_2)\\
\mathrm{semiring}(t, h) &= \perp \quad \text{(otherwise)}
\end{aligned}$$

As before, we consider an operational version of the denotational semantics of NRC over K-values. This is shown in Figure 23. As usual, there are two judgments, one for expression evaluation and one for iterating over a set. Many of the rules not involving collections are standard. The semiring function handles the cases for $\emptyset$, $\cup$, and $\{e\}$.

There is a mismatch between the denotational semantics on K-values and the operational semantics. The latter produces annotated stores, and we need to translate these to K-values in order to be able to relate the denotational and operational semantics. The desired translation is different from the ones we have needed so far. We define

$$\begin{aligned}
\sigma^{(h)} \Uparrow^K_{\mathsf{int}} l &= \sigma(l)\\
\sigma^{(h)} \Uparrow^K_{\mathsf{bool}} l &= \sigma(l)\\
\sigma^{(h)} \Uparrow^K_{\tau_1 \times \tau_2} l &= (\sigma^{(h)} \Uparrow^K_{\tau_1} l_1, \sigma^{(h)} \Uparrow^K_{\tau_1} l_1) \quad (\sigma(l) = (l_1, l_2))\\
\sigma^{(h)} \Uparrow^K_{\{\tau\}} l &= \lambda x. \sum \{h(l)(l')\\
&\qquad \mid l' \in \mathrm{dom}(\sigma(l)), \sigma^{(h)} \Uparrow^K_{\{\tau\}} l' = x\}
\end{aligned}$$

The translation steps for the basic types and pairing are straightforward. For collection types, we need to construct a K-collection corresponding to l; to do so, given an input x we sum together the values $h(l)(l')$ for each label l' in $\mathrm{dom}(\sigma(l))$ such that the K-value of l' in $\sigma^{(h)}$ is x. In particular, note that we *ignore* the multiplicity of l' in $\sigma(l)$ here.

We can now show the equivalence of the operational and denotational presentations of the semiring semantics:

$$\frac{}{\sigma^{(h)}, l \leftarrow t \Downarrow_K \sigma[l := t]^{(h[l:=\mathrm{semiring}(t,h)])}}$$

$$\frac{\sigma^{(h)}, l' \leftarrow e_1 \Downarrow_K \sigma'^{(h')} \quad \sigma'^{(h')}, l \leftarrow e_2[l'/x] \Downarrow_K \sigma''^{(h'')} \quad l' \text{ fresh}}{\sigma^{(h)}, l \leftarrow \text{let } x = e_1 \text{ in } e_2 \Downarrow_K \sigma''^{(h'')}}$$

$$\frac{\sigma(l') = (l_1, l_2)}{\sigma^{(h)}, l \leftarrow \pi_i(l') \Downarrow_K \sigma[l := \sigma(l_i)]^{(h[l:=h(l_i)])}}$$

$$\frac{\sigma(l') = b \quad \sigma^{(h)}, l \leftarrow e_b \Downarrow_K \sigma'^{(h')}}{\sigma^{(h)}, l \leftarrow \text{if } l' \text{ then } e_{\mathsf{t}} \text{ else } e_{\mathsf{f}} \Downarrow_K \sigma'^{(h')}}$$

$$\frac{\sigma^{(h)}, x \in \sigma(l')^{(h(l'))}, e \Downarrow^{\star}_K \sigma'^{(h')}, L'^{(k')}}{\sigma^{(h)}, l \leftarrow \bigcup\{e \mid x \in l'\} \Downarrow_K \sigma'[l := \bigsqcup \sigma'[L']]^{(h'[l:=k' \bullet_{\mathcal{K}} h'])}}$$

$$\frac{}{\sigma^{(h)}, x \in \emptyset^{(k)}, e \Downarrow^{\star}_K \sigma^{(h)}, \emptyset^{(0_{\mathcal{K}})}}$$

$$\frac{\sigma^{(h)}, x \in L_1^{(k)}, e \Downarrow^{\star}_K \sigma_1^{(h_1)}, L_1'^{(k_1)} \quad \sigma^{(h)}, x \in L_2^{(k)}, e \Downarrow^{\star}_K \sigma_2^{(h_2)}, L_2'^{(k_2)}}{\sigma^{(h)}, x \in (L_1 \oplus L_2)^{(k)}, e \Downarrow^{\star}_K (\sigma_1 \uplus_\sigma \sigma_2)^{(h_1 \uplus_h h_2)}, (L_1' \oplus L_2')^{(k_1 +_{\mathcal{K}} k_2)}}$$

$$\frac{\sigma^{(h)}, l' \leftarrow e[l/x] \Downarrow_K \sigma'^{(h')} \quad l' \text{ fresh}}{\sigma^{(h)}, x \in \{l : m\}^{(k)}, e \Downarrow^{\star}_K \sigma'^{(h')}, \{l' : m\}^{(k(l) \cdot \eta_{\mathcal{K}}(l'))}}$$

Figure 23. Semiring provenance, operationally

A	B	D	
1	2	7	$R_1 S_3$
1	3	7	$R_2 S_3$

A	D	
1	7	$R_1 S_3 + R_2 S_3$

Output table $Q_1(A, B, D)$ Output table $Q_3(A, D)$

Figure 24. Semiring provenance extraction examples

Theorem 12.

1. *Suppose $\Gamma \vdash e : \tau$ and $\Psi \vdash \sigma, \gamma : \Gamma$. Then $\sigma^{(h)}, l \leftarrow e \Downarrow_K \sigma'^{(h')}$ if and only if $K[\![e]\!](\sigma^{(h)} \Uparrow^K_\Gamma \gamma) = \sigma'^{(h')} \Uparrow^K_\tau l$.*
2. *Suppose $\Gamma, x : \tau \vdash e : \{\tau'\}$ and $\Psi \vdash \sigma, \gamma : \Gamma$. Then $\sigma^{(h)}, x \in L, e \Downarrow^{\star}_K \sigma'^{(h')}, L'$ if and only if $\{K[\![e]\!]\gamma[x := v] \mid v \in \sigma^{(h)} \Uparrow^K_{\{\tau\}} L\} = \sigma'^{(h')} \Uparrow^K_{\{\tau'\}} L'$.*

Our main result is that extraction semantics is correct with respect to the operational semantics:

Theorem 13. *1. If $\sigma, l \Leftarrow e \Downarrow \sigma', T$ then $\sigma^{(h)}, l \leftarrow e \Downarrow_K \sigma'^{(h')}$ holds if and only if $h, T \rightsquigarrow_K h'$.*
2. *If $\sigma, x \in L, e \Downarrow^{\star} \sigma', L', \Theta$ then $\sigma^{(h)}, x \in L^{(k)}, e \Downarrow^{\star}_K \sigma'^{(h')}, L'^{(k')}$ if and only if $h, k, \Theta \rightsquigarrow_K h', k'$.*

Example 6 Figure 24 shows the result of semiring-provenance extraction on Q_1. Here, we write R_1, S_1, etc. for the annotations of r_1 in r, s_1 in s, etc. respectively. The second query Q_2 involves $\sum$ expressions, which are not handled by the semiring model. Instead, the second part of Figure 24 shows the result of semiring provenance extraction on $Q_3 = \{(A : x.A, D : x.D) \mid x \in Q_1\}$, where we have merged the two copies of the record $(A : 1, D : 7)$ together and added their K-values.

5.4 Discussion

The fact that several distinct forms of provenance can all be extracted from traces is a clear qualitative indication that traces are very general. This generality is not surprising in light of the fidelity property, which essentially requires that the traces accurately repre-

$$\frac{}{h, l \leftarrow t \leadsto_K h[l := \text{semiring}(t, h)]} \qquad \frac{h, T_1 \leadsto_K h' \quad h', T_2 \leadsto_K h''}{h, T_1; T_2 \leadsto_K h''}$$

$$\frac{}{h, l \leftarrow \text{proj}_i(l', l_i) \leadsto_K h[l := h(l_i)]} \qquad \frac{h, T \leadsto_K h'}{h, \text{cond}_l(l', b, T) \leadsto_K h'}$$

$$\frac{h, h(l'), \Theta \leadsto_K^* h', k'}{h, l \leftarrow \text{comp}(l', \Theta) \leadsto_K h'[l := k' \bullet_K h']}$$

$$\frac{}{h, k, \emptyset \leadsto_K h, 0_K} \qquad \frac{h, k, \Theta_1 \leadsto_K h_1, k_1 \quad h, k, \Theta_2 \leadsto_K h_2, k_2}{h, k, \Theta_1 \oplus \Theta_2 \leadsto_K h_1 \uplus_h h_2, k_1 +_K k_2}$$

$$\frac{h, T \leadsto_K h'}{h, k, \{[l]T : m\} \leadsto_K^* h', k(l) \cdot_K \eta_K(\text{out}(T))}$$

Figure 25. Extracting semiring provenance

sent the query in all inputs. In fact, the provenance extraction rules do not inspect the expression annotations $x.e, e_1, e_2$ in comprehension and conditional traces; thus, they all work correctly even without these annotations. Also, the extraction rules do not have access to the underlying store σ; nor do they need to reconstruct the intermediate store. The trace itself records enough information about the store labels actually accessed.

6. Trace slicing

As noted above, traces are often large. Traces are also difficult to interpret because they reduce computations to very basic steps, like machine code. In this section, we consider *slicing* and other simplifications for making trace information more useful and readable. However, formalizing these techniques appears nontrivial, and is beyond the scope of this paper. Here we only consider examples of trace slicing and simplification techniques that discard some of the details of the trace information to make it more readable.

Example 7 Recall query Q_1. If we are only interested in how row l_1 in the output was computed, then the following *backwards trace slice* answers this question.

```
l <- comp(r,{
  [r1] x11 <- proj_C(r1,r13); x1 <- comp(s,{
    [s3] x131 <- proj_C(s3,s31); x132 <- x11 = x131;
         cond(x132,t,l11 <- proj_A(r1,r11);
                    l12 <- proj_B(r1,r12);
                    l13 <- proj_D(s3,s32);
                    l1 <- (A:l11,B:l12,D:l13);
                    x136 <- {l1})})})
```

Note that the slice refers only to the rows r_1 and s_3 that contribute to the semiring-provenance of l_1. Moreover, the where-provenance and dependency-provenance of l_1, l_{11}, l_{12}, and l_{13} can be extracted from this slice.

To make the slice more readable, we can discard information about projection and assignment steps and substitute expressions for labels:

```
l <- comp(r,{
  [r1] x1 <- comp(s,{
    [s3] cond(r13 = s31,t,l1 <- (A:r11,B:r12,D:s32);
                         x136 <- {l1})})})
```

We can further simplify this to an expression $\{(A : r_{11}, B : r_{12}, D : s_{32})\}$ that shows how to calculate l_1 from the original input, but this is not guaranteed to be valid if the input is changed.

Example 8 In query Q_2, if we are only interested in the value 7 labeled by l'_{12}, its (simplified) backwards trace slice is:

```
l12' <- sum(s,{[s1] cond(s11 = 2, t, x13 <- s12),
```

```
[s2] cond(s12 = 2, t, x23 <- s22),
[s3] cond(s13 = 2, f, x33 <- 0)});
```

and from this we can extract an expression such as $s_{12} + s_{22}$ that describes how the result was computed.

7. Related and future work

Provenance has been studied for database queries under various names, including "source tagging" and "lineage". We have already discussed where-provenance, dependency provenance and the semiring model. Wang and Madnick (1990) described an early provenance semantics meant to capture the original and intermediate sources of data in the result of a query. Cui, Widom and Wiener defined *lineage*, which aims to identify source data relevant to part of the output. Buneman et al. (2001) also introduced *why–provenance*, which attempts to highlight parts of the input that explain why a part of the output is the way it is. As discussed earlier, lineage and why-provenance are instances of the semiring model. Recently, Benjelloun et al. (2006) have studied a new form of lineage in the Trio system. According to Green (personal communication), Trio's lineage model is also an instance of the semiring model, so can also be extracted from traces.

Buneman et al. (2006) and Buneman et al. (2007) investigated provenance for database updates, an important scenario because many scientific databases are *curated*, or maintained via frequent manual updates. Provenance is essential for evaluating the scientific value of curated databases (Buneman et al. 2008). We have not considered traces for update languages in this paper. This is an important direction for future work.

Provenance has also been studied in the context of *(scientific) workflows*, that is, high-level visual programming languages and systems developed recently as interfaces to complex distributed Grid computation. Techniques for workflow provenance are surveyed by Bose and Frew (2005) and Simmhan et al. (2005). Most such systems essentially record call graphs including the names and parameters of macroscopic computation steps, input and output filenames, and other system metadata such as architecture, operating system and library versions. Similarly, provenance-aware storage systems (Muniswamy-Reddy et al. 2006) record high-level trace information about files and processes, such as the files read and written by a process.

To our knowledge formal semantics have not been developed for most workflow systems that provide provenance tracking. Many of them involve concurrency so defining their semantics may be nontrivial. One well-specified approach is the NRC-based "dataflow" model of (Hidders et al. 2007), who define an instrumented semantics that records "runs" and consider extracting provenance from runs. However, their formalization is incomplete and does not examine semantic correctness properties comparable to consistency and fidelity; moreover, they have not established the exact relationship between their runs and existing forms of provenance.

As discussed in the introduction, provenance traces are related to the traces used in the adaptive functional programming language AFL (Acar et al. 2006). The main difference is that AFL traces are meant to model efficient self-adjusting computation implementations, whereas provenance traces are intended as a model of execution history that can be used to answer high-level queries comparable to other provenance models. Nevertheless, efficiency is obviously an important issue for provenance-tracking techniques. The problem of efficiently recomputing query results after the input changes, also called *view maintenance*, has been studied extensively for *materialized views* (cached query results) in relational databases (Gupta and Mumick 1995). View maintenance does not appear to have been studied in general for NRC, but provenance

traces may provide a starting point for doing so. View maintenance in the presence of provenance seems to be an open problem.

Provenance traces may also be useful in studying the *view update* problem for NRC queries, that is, the problem of updating the input of a query to accommodate a desired change to the output. This is closely related to bidirectional computation techniques that have been developed for XML trees (Foster et al. 2007), flat relational queries (Bohannon et al. 2006), and text processing (Bohannon et al. 2008). Provenance-like metadata has already been found useful in some of this work. Thus, we believe that it will be worthwhile to further study the relationship between provenance traces and bidirectional computation.

There is a large body of related work on dynamic analysis techniques, including slicing, debugging, justification, information flow, dependence tracking, and profiling techniques, in which execution traces play an essential role. We cannot give a comprehensive overview of this work here, but refer to (Venkatesh 1991; Arora et al. 1993; Abadi et al. 1996; Field and Tip 1998; Abadi et al. 1999; Ochoa et al. 2004) as sources we found useful for inspiration. However, to our knowledge, none of these techniques have been studied in the context of database query languages, and our work reported previously in (Cheney et al. 2007) and in this paper is the first to connect any of these topics to provenance.

Trace semantics is also employed in static analysis; in particular, see (Rival and Mauborgne 2007). Cheney et al. (2007) defined a type-and-effect-style static analysis for dependency provenance; to our knowledge, there is no other prior work on using static analysis to approximate provenance or optimize dynamic provenance tracking.

8. Conclusions

Provenance is an important topic in a variety of settings, particularly where computer systems such as databases are being used in new ways for scientific research. The semantic foundations of provenance, however, are not well understood. This makes it difficult to judge the correctness and effectiveness of existing proposals and to study their strengths and weaknesses.

This paper develops a foundational approach based on *provenance traces*, which can be viewed as explanations of the operational behavior of a query not on just the current input but also on other possible (well-defined) inputs. We define and give traced operational semantics and adaptation semantics for traces and prove *consistency* and *fidelity* properties that characterize precisely how traces produced by our approach record the run-time behavior of queries. The proof of fidelity, in particular, involves subtleties not evident in other trace semantics systems such as AFL (Acar et al. 2006) due to the presence of collection types and comprehensions, which are characteristic of database query languages.

Provenance traces are very general, as illustrated by the fact that other forms of provenance information may be extracted from them. For instance, we show how to extract where-provenance, dependency provenance, and semiring provenance from traces. Depending on the needs of the application, these specialized forms of provenance may be preferable to provenance traces due to efficiency concerns. As a further application, we informally discuss how we may slice or simplify traces to extract smaller traces that are more relevant to part of the input or output.

To our knowledge, our work is the first to formally investigate trace semantics for collection types or database query languages and the first to relate traces to other models of provenance in databases. There are a number of compelling directions for future work, including formalizing interesting definitions of trace slices, developing efficient techniques for generating and querying provenance traces, and relating provenance traces to the view-maintenance and view-update problems.

References

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

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

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, 2006.

Tarun Arora, Raghu Ramakrishnan, William G. Roth, Praveen Seshadri, and Divesh Srivastava. Explaining program execution in deductive systems. In *Deductive and Object-Oriented Databases*, pages 101–119, 1993.

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

Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: a language for updatable views. In *PODS*, pages 338–347. ACM Press, 2006.

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *POPL*, pages 407–419. ACM, 2008.

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*, number 1973 in LNCS, pages 316–330. Springer, 2001.

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

Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. In *ICDT*, number 4353 in LNCS, pages 209–223. Springer, 2007.

Peter Buneman, James Cheney, Wang-Chiew Tan, and Stijn Vansummeren. Curated databases. In *PODS*, pages 1–12, 2008.

James Cheney, Amal Ahmed, and Umut A. Acar. Provenance as dependency analysis. In *DBPL*, volume 4797 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2007.

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.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

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.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.

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

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

Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.

Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. A formal model of dataflow repositories. In *DILS*, volume 4544 of *LNCS*, pages 105–121. Springer, 2007.

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

Claudio Ochoa, Josep Silva, and Germán Vidal. Dynamic slicing based on redex trails. In *PEPM*, pages 123–134. ACM Press, 2004.

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, New York, NY, USA, 2008. ACM.

Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5):26, 2007.

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

G. A. Venkatesh. The semantic approach to program slicing. In *PLDI*, pages 107–119. ACM Press, 1991.

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.