Database Queries that Explain their Work

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

Amal Ahmed
Northeastern University
amal@ccs.neu.edu

Umut A. Acar
Carnegie Mellon University &
INRIA-Rocquencourt
umut@cs.cmu.edu

Abstract

Provenance for database queries or scientific workflows is often motivated as providing *explanation*, increasing understanding of the underlying data sources and processes used to compute the query, and *reproducibility*, the capability to recompute the results on different inputs, possibly specialized to a part of the output. Many provenance systems claim to provide such capabilities; however, most lack formal definitions or guarantees of these properties, while others provide formal guarantees only for relatively limited classes of changes. Building on recent work on provenance traces and slicing for functional programming languages, we introduce a detailed tracing model of provenance for multiset-valued Nested Relational Calculus, define trace slicing algorithms that extract subtraces needed to explain or recompute specific parts of the output, and define query slicing and differencing techniques that support explanation. We state and prove correctness properties for these techniques and present a proof-of-concept implementation in Haskell.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords provenance, database queries, slicing

1. Introduction

Over the past decade, the use of complex computer systems in science has increased dramatically: databases, scientific workflow systems, clusters, and cloud computing based on frameworks such as MapReduce [17] or PigLatin [27] are now routinely used for scientific data analysis. With this shift to computational science based on (often) unreliable components and noisy data comes decreased transparency, and an increased need to understand the results of complex computations by auditing the underlying processes.

This need has motivated work on *provenance* in databases, scientific workflow systems, and many other settings [7, 26]. There is now a great deal of research on extending such systems with rich provenance-tracking features. Generally, these systems aim to provide high-level explanations intended to aid the user in understanding how a computation was performed, by recording and presenting additional "trace" information.

PPDP '14, September 08–10, 2014, Canterbury, United Kingdom.

ACM 978-1-4503-2947-7/14/09. . . $15.00.
http://dx.doi.org/10.1145/2643135.2643143

Over time, two distinct approaches to provenance have emerged: (1) the use of *annotations* propagated through database queries to illustrate *where-provenance* linking results to source data [7], *lineage* or *why-provenance* linking result records to sets of witnessing input records [16], or *how-provenance* describing how results were produced via algebraic expressions [21], and (2) the use of graphical *provenance traces* to illustrate how workflow computations construct final results from inputs and configuration parameters [5, 22, 30]. However, to date few systems formally specify the semantics of provenance or give formal guarantees characterizing how provenance "explains" results.

For example, scientists often conduct parameter sweeps to search for interesting results. The provenance trace of such a computation may be large and difficult to navigate. Once the most promising results have been identified, a scientist may want to extract just that information that is needed to explain the result, without showing all of the intermediate search steps or uninteresting results. Conversely, if the results are counterintuitive, the scientist may want to identify the underlying data that contributed to the anomalous result. Missier et al. [25] introduced the idea of a "golden trail", or a subset of the provenance trace that explains, or allows reproduction of, a high-value part of the output. They proposed techniques for extracting "golden trails" using recursive Datalog queries over provenance graphs; however, they did not propose definitions of correctness or reproducibility.

It is a natural question to ask how we know when a proposed solution, such as Missier et al.'s "golden trail" queries, correctly explains or can be used to correctly reproduce the behavior of the original computation. It seems to have been taken for granted that simple graph traversals suffice to at least overapproximate the desired subset of the graph. As far as we know, it is still an open question how to define and prove such correctness properties for most provenance techniques. In fact, these properties might be defined and formalized in a number of ways, reflecting different modeling choices or requirements. In any case, in the absence of clear statements and proofs of correctness, claims that different forms of provenance "explain" or allow "reproducibility" are difficult to evaluate objectively.

The main contribution of this paper is to formalize and prove the correctness of an approach to fine-grained provenance for database queries. We build on our approach developed in prior work, which we briefly recapitulate. Our approach is based on analogies between the goals of provenance tracking for databases and workflows, and those of classical techniques for program comprehension and analysis, particularly *program slicing* [32] and *information flow* [29]. Both program slicing and information flow rely critically on notions of *dependence*, such as the familiar control-flow and data-flow dependences in programming languages.

We previously introduced a provenance model for NRC (including difference and aggregation operations) called *dependency provenance* [12], and showed how it can be used to compute *data*

slices, that is, subsets of the input to the query that include all of the information relevant to a selected part of the output. Some other forms of provenance for database query languages, such as how-provenance [21], satisfy similar formal guarantees that can be used to predict how the output would change under certain classes of input changes, specifically those expressible by semiring homomorphisms. For example, Amsterdamer et al.'s system [3] is based on the semiring provenance model, so the effects of deletions on parts of the output can be predicted by inspecting their provenance, but other kinds of changes are not supported.

More recently, we proposed an approach to provenance called *self-explaining computation* [11] and explored it in the context of a general-purpose functional programming language [1, 28]. In this approach, detailed execution traces are used as a form of provenance. Traces explain results in the sense that they can be *replayed* to recompute the results, and they can be *sliced* to obtain smaller traces that provide more concise explanations of parts of the output. Trace slicing also produces a slice of the input showing what was needed by the trace to compute the output. Moreover, other forms of provenance can be extracted from traces (or slices), and we also showed that traces can be used to compute program slices efficiently through lazy evaluation. Finally, we showed how traces support *differential slicing* techniques that can highlight the differences between program runs in order to explain and precisely localize bugs in the program or errors in the input data.

Our long-term vision is to develop self-explaining computation techniques covering all components used in day-to-day scientific practice. Databases are probably the single most important such component. Since our previous work already applies to a general-purpose programming language, one way to proceed would be to simply implement an interpreter for NRC in this language, and inherit the slicing behavior from that. However, without some further inlining or optimization, this naive strategy would yield traces that record both the behavior of the NRC query and its interpreter, along with internal data structures and representation choices whose details are (intuitively) irrelevant to understanding the high-level behavior of the query.

1.1 Technical overview

In this paper, we develop a tracing semantics and trace slicing techniques tailored to NRC (over a multiset semantics). This semantics evaluates a query Q over an input database (i.e. environment γ mapping relation names to table values), yielding the usual result value v as well as a *trace* T. Traces are typically large and difficult to decipher, so we consider a scenario where a user has run Q, inspected the results v, and requests an explanation for a part of the result, such as a field value of a single record. As in our previous work for functional programs, we use partial values with "holes" $\square$ to describe parts of the output that are to be explained. For example, if the result of a program is just a pair $(1, 2)$ then the pattern $(1, \square)$ can be used to request an explanation for just the first component. Given a partial value p matching the output, our approach computes a "slice" consisting of a partial trace and a partial input environment, where components not necessary for recomputing the explained output part p have been deleted.

The main technical contribution of this paper over our previous work [1, 28] is its treatment of tracing and slicing for collections. There are two underlying technical challenges; we illustrate both (and our solutions) via a simple example query $Q = \sigma_{A<B}(R) \cup \rho_{A\mapsto B, B\mapsto A}(\sigma_{A\geq B}(R))$ over a table R with attributes A, B. Here, σ_ϕ is relational selection of all tuples satisfying a predicate ϕ and $\rho_{A\mapsto B, B\mapsto A}$ is renaming. Thus, Q simply swaps the fields of records where $A \geq B$, and leaves other records alone.

The first challenge is how to address elements of multisets reliably across different executions and support propagation of addresses in the output backwards towards the input. Our solution is to use a mildly enriched semantics in which multiset elements carry explicit labels; that is, we view multisets of elements from X as functions $I \to X$ from some index set to X. For example, if R is labeled as follows and we use this enriched semantics to evaluate the above query Q on R, we get a result:

$$R = \begin{array}{c|ccc} id & A & B & C \\ \hline [r_1] & 1 & 2 & 7 \\ [r_2] & 2 & 3 & 8 \\ [r_3] & 4 & 3 & 9 \end{array} \qquad Q(R) = \begin{array}{c|ccc} id & A & B & C \\ \hline [1, r_1] & 1 & 2 & 7 \\ [1, r_2] & 2 & 3 & 8 \\ [2, r_3] & 3 & 4 & 9 \end{array}$$

where in each case the id column contains a distinct index r_i. In $Q(R)$, the first '1' or '2' in each index indicates whether the row was generated by the left or right subexpression in the union ('$\cup$'). In general, we use sequences of natural numbers $[i_1, \ldots, i_n] \in \mathbb{N}^*$ as indices, and we maintain a stronger invariant: the set of indexes used in a multiset must form a *prefix code*. We define a semantics for NRC expressions over such collections that is fully deterministic and does not resort to generation of fresh intermediate labels; in particular, the union operation adjusts the labels to maintain distinctness.

The second technical challenge involves extending *patterns* for partial collections. In our previous work, any subexpression of a value can be replaced by a hole. This works well in a conventional functional language, where typical values (such as lists and trees) are essentially initial algebras built up by structural induction. However, when we consider unordered collections such as bags, our previous approach becomes awkward.

For example, if we want to use a pattern to focus on only the B field of the second record in query result $Q(R)$, we can only do this by deleting the other record values, and the smallest such pattern is $\{[1, r_1].\square, [2, r_2].\langle A{:}\square, B{:}3, C{:}\square\rangle, [2, r_3].\square\}$. This is tolerable if there are only a few elements, but if there are hundreds or millions of elements and we are only interested in one, this is a significant overhead. Therefore, we introduce *enriched patterns* that allow us to replace entire subsets with holes, for example, $p' = \{[2, r_2].\langle B{:}3; \square\rangle\} \dot\cup \square$. Enriched patterns are not just a convenience; we show experimentally that they allow traces and slices to be smaller (and computed faster) by an order of magnitude or more.

1.2 Outline

The rest of this paper is structured as follows. Section 2 reviews the (multiset-valued) Nested Relational Calculus and presents our tracing semantics and deterministic labeling scheme. Section 3 presents trace slicing, including a simple form of patterns. Section 4 shows how to enrich our pattern language to allow for partial record and partial set patterns, which considerably increase the expressiveness of the pattern language, leading to smaller slices. Section 5 presents the query slicing algorithm and shows how to compute differential slices. Section 6 presents additional examples and discussion. Section 7 presents our implementation demonstrating the benefits of laziness and enriched patterns for trace slicing. Section 8 discusses related and future work and Section 9 concludes.

Due to space limitations, and in order to make room for examples and high-level discussion, some (mostly routine) formal details and proofs are relegated to a companion technical report [13].

2. Traced Evaluation for NRC

The nested relational calculus [9] is a simply-typed core language with collection types that can express queries on nested data similar to those of SQL on flat relations, but has simpler syntax and cleaner semantics. In this section, we show how to extend the ideas and machinery developed in our previous work on traces and slicing for functional languages [28] to NRC. Developing formal foundations

$$
\begin{array}{lll}
\text{Operations} & \mathtt{f} ::= & + \mid - \mid * \mid / \mid = \mid < \mid \leq \mid \cdots \\
\text{Expressions} & e ::= & \mathtt{c} \mid \mathtt{f}(e_1, \ldots, e_n) \mid x \mid \mathtt{let}\ e = x\ \mathtt{in}\ e' \\
& \mid & \langle A_1 : e_1, \ldots, A_n : e_n \rangle \mid e.A \mid \mathtt{if}(e, e', e'') \\
& \mid & \emptyset \mid \{e\} \mid e_1 \cup e_2 \mid \bigcup\{e' \mid x \in e\} \\
& \mid & \mathtt{empty}\ e \mid \mathtt{sum}\ e \mid \cdots \\
\text{Labels} & \ell ::= & \ell.\ell' \mid \epsilon \mid n \\
\text{Values} & v ::= & \mathtt{c} \mid \langle A_1 : v_1, \ldots A_n : v_n \rangle \\
& \mid & \emptyset \mid \{\ell_1.v_1, \ldots, \ell_n.v_n\} \\
\text{Environments} & \gamma ::= & [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \\
\text{Traces} & T ::= & \cdots \mid \mathtt{if}(T, e', e'') \triangleright_{\mathtt{true}} T \mid \mathtt{if}(T, e', e'') \triangleright_{\mathtt{false}} T \\
& \mid & \bigcup\{e \mid x \in T\} \triangleright \Theta \\
\text{Trace Sets} & \Theta ::= & \{\ell_1.T_1, \ldots, \ell_n.T_n\} \\
\text{Types} & \tau ::= & \mathtt{int} \mid \mathtt{bool} \mid \langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle \mid \{\tau\} \\
\text{Type Contexts} & \Gamma ::= & x_1 : \tau_1, \ldots, x_n : \tau_n
\end{array}
$$

Figure 1. NRC expressions, values, traces, and types.

for tracking provenance in the presence of unordered collections presents a number of challenges not encountered in the functional programming setting, as we will explain.

2.1 Syntax and Dynamic Semantics

Figure 1 presents the abstract syntax of NRC expressions, values, and traces. The expression $\emptyset$ denotes the empty collection, $\{e\}$ constructs a singleton collection, and $e_1 \cup e_2$ takes the (multiset) union of two collections. The operation $\mathtt{sum}\ e$ computes the sum of a collection of integers, while the predicate $\mathtt{empty}\ e$ tests whether the collection denoted by e is empty. Additional aggregation operations such as count, maximum and average can easily be accommodated. Finally, the comprehension operation $\bigcup\{e' \mid x \in e\}$ iterates over the collection obtained by evaluating e, evaluating $e'(x)$ with x bound to each element of the collection in turn, and returning a collection containing the union of all of the results. We sometimes consider pairs (e_1, e_2), a special case of records $\langle \#_1 : e_1, \#_2 : e_2 \rangle$ using two designated field names $\#_1$ and $\#_2$. Many trace forms are similar to those for expressions; only the differences are shown.

Labels are sequences $\ell = [i_i, \ldots, i_n] \in \mathbb{N}^*$, possibly empty. The empty sequence is written ϵ, and labels can be concatenated $\ell \cdot \ell'$; concatenation is associative. Record field names are written $A, B, A_1, A_2, \ldots$.

Values in NRC include constants $\mathtt{c}$, which we assume include at least booleans and integers. Record values are essentially partial functions from field names to values, written $\langle A_1 : v_1, \ldots, A_n : v_n \rangle$. Collection values are essentially partial, finite-domain functions from labels in $\mathbb{N}^*$ to values, which we write $\{\ell_1.v_1, \ldots, \ell_n.v_n\}$. Since they denote functions, collections and records are identified up to reordering of their elements, and their field names or labels are always distinct. We write $\ell \cdot v$ for the operation that prepends ℓ to each of the labels in a set v, that is,

$$\ell \cdot \{\ell_1.v_1, \ldots, \ell_n.v_n\} = \{\ell \cdot \ell_1.v_1, \ldots, \ell \cdot \ell_n.v_n\}\ .$$

Other operations on labels and labeled collections will be introduced in due course.

The labels on the elements of a collection provide us with a persistent address for a particular element of the collection. This capability is essential when asking and answering provenance queries about parts of the source or output data, and when tracking fine-grained dependencies.

Both expressions and traces are subject to a type system. NRC types include collection types $\{\tau\}$ which are often taken to be sets, bags (multisets), or lists, though in this paper, we consider multiset collections only. However, types do not play a significant role in this paper so the typing rules are omitted. For expressions, the typing judgment $\Gamma \vdash e : \tau$ is standard and the typing rules for trace well-formedness $\Gamma \vdash T : \tau$ are presented in the companion technical report [13].

Traced evaluation NRC traces include a trace form corresponding to each of the expressions described above. The structure of the traces is best understood by inspecting the traced evaluation rules (Figure 2), which define a judgment $\gamma, e \Downarrow v, T$ indicating that evaluating an expression e in environment γ yields a value v and a trace T. We assume an environment Σ associating constants and function symbols with their types, and write $\hat{\mathtt{f}}$ or $\hat{+}$ for the semantic operations corresponding to f or $+$, and so on. In most cases, the trace form is similar to the expression form; for example the trace of a constant or variable is a constant trace $\mathtt{c}$, the trace of a primitive operation $\mathtt{f}(e_1, \ldots, e_n)$ is a primitive operation trace $\mathtt{f}(T_1, \ldots, T_n)$ applied to the traces T_i of the arguments e_i, the trace of a record expression is a trace record constructor $\langle A_1 : T_1, \ldots, A_n : T_n \rangle$, and the trace of a field projection $e.A$ is a trace $T.A$. Also, the trace of a let-binding is a let-binding trace $\mathtt{let}\ x = T_1\ \mathtt{in}\ T_2$, where x is bound in T_2. In these cases, the traces mimic the expression structure.

The traced evaluation rules for conditionals illustrate that traces differ from expressions in recording control flow decisions. The trace of a conditional is a conditional trace $\mathtt{if}(T, e_1, e_2) \triangleright_b T'$ where T is the trace of the conditional test, b is the Boolean value of the test e_1, and T' is the trace of the taken branch. The expressions e_1 and e_2 are not strictly necessary but retained to preserve structural similarity to the original expression.

The trace of $\emptyset$ is a constant trace $\emptyset$. To evaluate a singleton-collection constructor $\{e\}$, we evaluate e to obtain a value v and return the singleton $\{\epsilon.v\}$ with empty label ϵ. We return the singleton trace $\{T\}$ recording the trace for the evaluation of the element. To evaluate the union of two expressions, we evaluate each one and take the semantic union (written $\uplus$) of the resulting collections, with a '1' or '2' concatenated onto the beginning of each label to reflect whether each element came from the first or second part of the union; the union trace $T_1 \cup T_2$ records the traces for the evaluation of the two subexpressions. For $\mathtt{sum}\ e$, evaluating e yields a collection of numbers whose sum we return, together with a sum trace $\mathtt{sum}\ T$ recording the trace for evaluation of e. Evaluation of emptiness tests $\mathtt{empty}\ e$ is analogous, yielding a trace $\mathtt{empty}\ T$.

To evaluate a comprehension $\bigcup\{e' \mid x \in e\}$, we first evaluate e, which yields a collection v and trace T, and then (using auxiliary judgment $\gamma, x \in v, e' \Downarrow^* v', \Theta$) evaluate e' repeatedly with x bound to each element v_i of the collection v to get resulting values v'_i and corresponding traces T'_i. We return a new collection $v' = \{\ell_1 \cdot v'_1, \ldots, \ell_n \cdot v'_n\}$; similarly we return a labeled set of traces $\Theta = \{\ell_1.T_1, \ldots, \ell_n.T_n\}$. (Analogously to values, trace sets are essentially finite partial functions from labels to traces). For each of these collections, we prepend the appropriate label ℓ_i of the corresponding input element.

A technical point of note is that the resulting trace T_i may contain free occurrences of x. As in our trace semantics for functional programs, these variables serve as markers in T_i that will be critical for the trace replay semantics. The comprehension trace records, using the notation $\bigcup\{e' \mid x \in T\} \triangleright \{\ell_1.T_1, \ldots, \ell_n.T_n\}$, that the trace T was used to compute a multiset v, and x was bound to each element $\ell_i.v_i$ in v in turn, with trace T_i showing how the corresponding subset of the result was computed. The comprehension trace also records the expression e' and bound variable x, which are again not strictly necessary but preserve the structural similarity to the original expression.

At this point it is useful to provide some informal motivation for the labeling semantics, compared for example to other semantics that use annotations or labels as a form of provenance. We do

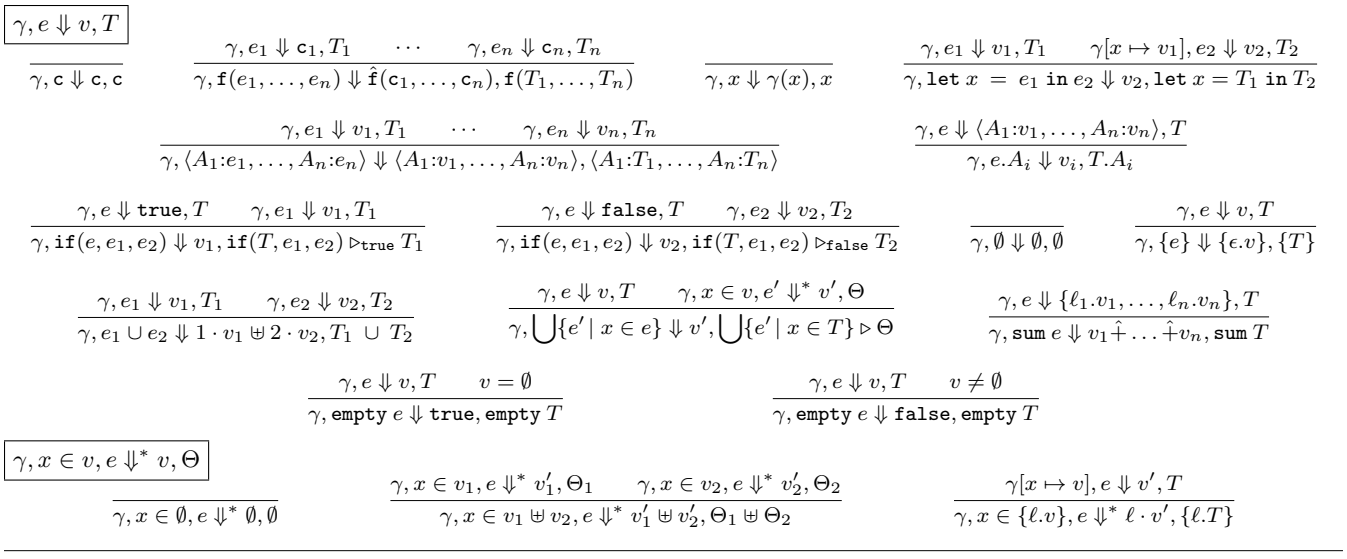

Figure 2. Traced evaluation.

not view the labels themselves as provenance; instead, they provide a useful infrastructure for traces, which do capture a form of provenance. Moreover, by calculating the label of each part of an intermediate or final result deterministically (given the labels on the input), we provide a way to reliably refer to parts of the output, which otherwise may be unaddressable in a multiset-valued semantics. This is essential for supporting compositional slicing for operations such as let-binding or comprehension, where the output of one subexpression becomes a part of the input for another.

A central point of our semantics is that evaluation preserves the property that labels uniquely identify the elements of each multiset. This naturally assumes that the labels on the input collections are distinct. In fact, a stronger property is required: evaluation preserves the property that set labels form a *prefix code*. In the following, we write $x \leq y$ to indicate that sequence x is a prefix of sequence y.

Definition 2.1. *A* prefix code *over Σ is a set of sequences $L \subseteq \Sigma^*$ such that for every $x, y \in L$, if $x \leq y$ then $x = y$. A* sub-prefix code *of a prefix code L is a prefix code L' such that for all $x \in L$ there exists $y \in L'$ such that $y \leq x$. We write $L' \leq L$ to indicate that L' is a sub-prefix code of L. We say that L and L' are* prefix-disjoint *when no element of L is a prefix of an element of L' and vice versa.*

Let v be a collection $v = \{\ell_1.v_1, \ldots, \ell_n.v_n\}$. We define the domain of v to be $\mathrm{dom}(v) = \{\ell_1, \ldots, \ell_n\}$. . We say that a value or value environment is *prefix-labeled* if for every collection v occurring in it, the labels $\ell_1, \ldots, \ell_n$ are distinct and $\mathrm{dom}(v)$ is a prefix code. Similarly, we say that a trace is prefix-labeled if every labeled trace set $\Theta = \{\ell_1.T_1, \ldots, \ell_n.T_n\}$ is prefix-labeled.

Theorem 2.2. *If γ is prefix-labeled and $\gamma, e \Downarrow v, T$ then v and T are both prefix-labeled. Moreover, if γ and v are prefix-labeled and $\gamma, x \in v, e \Downarrow^* v', \Theta$ then v' and Θ are prefix-labeled, and in addition* $\mathrm{dom}(\Theta) = \mathrm{dom}(v) \leq \mathrm{dom}(v')$.

Proof. By induction on derivations. The key cases are those for union, which is straightforward, and for comprehensions. For the latter case we need the second part, to show that whenever γ and v are prefix-labeled, if $\gamma, x \in v, e \Downarrow^* v', \Theta$ then v' and Θ are prefix-labeled, and in addition $\mathrm{dom}(\Theta) = \mathrm{dom}(v)$ and $\mathrm{dom}(v)$ is a sub-prefix code of $\mathrm{dom}(v')$. □

The prefix code property is needed later in the slicing algorithms, when we will need it to match elements of collections produced by comprehensions with corresponding elements of the trace set Θ. From now on, we assume that all values, environments, and traces are prefix-labeled, so any labeled set is assumed to have the prefix code property.

We will use the following query as a running example.

$$Q = \bigcup\{\mathtt{if}(x.B = 3, \{\langle A{:}x.A,\, B{:}x.C\rangle\}, \{\}) \mid x \in R\}$$

This is a simple selection query; it identifies records in R that have B-value of 3, and returns record $\langle A{:}x.A, B{:}x.C\rangle$ containing x's A value and its C value renamed to B. The result of Q on the input R in the introduction is $Q(R) = \{[r_2].\langle A{:}2, B{:}8\rangle, [r_3].\langle A{:}4, B{:}9\rangle\}$, and the trace is:

$$T = \bigcup\{\_ \mid x \in R\} \triangleright \{\quad [r_1].\mathtt{if}(x.B = 3, \_, \_) \triangleright_{\mathtt{false}} \{\},\ [r_2].\mathtt{if}(x.B = 3, \_, \_) \triangleright_{\mathtt{true}} \{\_\},\ [r_3].\mathtt{if}(x.B = 3, \_, \_) \triangleright_{\mathtt{true}} \{\_\}\ \}$$

where $\_$ indicates omitted (easily inferrable) subexpressions.

Replay We introduce a judgment $\gamma, T \curvearrowright v$ for *replaying* a trace on a (possibly different) environment γ. The rules for replaying NRC traces are presented in Figure 3. Many of the rules are straightforward or analogous to the corresponding evaluation rules. Here we only discuss the replay rules for conditional and comprehension traces.

For the conditional rules, the basic idea is as follows. If replaying the trace T of the test yields the same boolean value b as recorded in the trace, we replay the trace of the taken branch. If the test yields a different value, then replay fails.

To replay a comprehension trace $\bigcup\{e \mid x \in T\} \triangleright \Theta$, rule RCOMP first replays trace T to update the set of elements v over which we will iterate. We define a separate judgment $\gamma, x \in v, \Theta \curvearrowright^* v'$ to iterate over set v and replay traces on the corresponding elements. For elements $\ell_i \in \mathrm{dom}(\Theta)$, we replay the corresponding trace $\Theta(\ell_i)$. Replay fails if v contains any labels not present in Θ.

Replaying a trace can fail if either the branch taken in a conditional test differs from that recorded in the trace, or the intermediate set obtained from rerunning a comprehension trace includes values whose labels are not present in the trace. This means, in particular, that changes to the input can be replayed if they only change base

$\boxed{\gamma, T \curvearrowright v}$

$$\frac{}{\gamma, \mathtt{c} \curvearrowright \mathtt{c}} \qquad \frac{\gamma, T_1 \curvearrowright \mathtt{c}_1 \quad \cdots \quad \gamma, T_n \curvearrowright \mathtt{c}_n}{\gamma, \mathtt{f}(T_1, \ldots, T_n) \curvearrowright \hat{\mathtt{f}}(\mathtt{c}_1, \ldots, \mathtt{c}_n)}$$

$$\frac{}{\gamma, x \curvearrowright \gamma(x)} \qquad \frac{\gamma, T_1 \curvearrowright v_1 \quad \gamma[x \mapsto v_1], T_2 \curvearrowright v_2}{\gamma, \mathtt{let}\ x\ =\ T_1\ \mathtt{in}\ T_2 \curvearrowright v_2}$$

$$\frac{\gamma, T_1 \curvearrowright v_1 \quad \cdots \quad \gamma, T_n \curvearrowright v_n}{\gamma, \langle A_1{:}T_1, \ldots, A_n{:}T_n\rangle \curvearrowright \langle A_1{:}v_1, \ldots, A_n{:}v_n\rangle}$$

$$\frac{\gamma, T \curvearrowright \langle A_1{:}v_1, \ldots, A_n{:}v_n\rangle}{\gamma, T.A_i \curvearrowright v_i} \qquad \frac{\gamma, T \curvearrowright \mathtt{true} \quad \gamma, T_1 \curvearrowright v_1}{\gamma, \mathtt{if}(T, e_1, e_2) \triangleright_{\mathtt{true}} T_1 \curvearrowright v_1}$$

$$\frac{\gamma, T \curvearrowright \mathtt{false} \quad \gamma, T_2 \curvearrowright v_2}{\gamma, \mathtt{if}(T, e_1, e_2) \triangleright_{\mathtt{false}} T_2 \curvearrowright v_2} \qquad \frac{}{\gamma, \emptyset \curvearrowright \emptyset} \qquad \frac{\gamma, T \curvearrowright v}{\gamma, \{T\} \curvearrowright \{\epsilon.v\}}$$

$$\frac{\gamma, T_1 \curvearrowright v_1 \quad \gamma, T_2 \curvearrowright v_2}{\gamma, T_1\ \cup\ T_2 \curvearrowright 1 \cdot v_1 \uplus 2 \cdot v_2} \qquad \frac{\gamma, T \curvearrowright \{\ell_1.v_1, \ldots, \ell_n.v_n\}}{\gamma, \mathtt{sum}\ T \curvearrowright v_1 \hat{+} \ldots \hat{+} v_n}$$

$$\frac{\gamma, T \curvearrowright v \quad v = \emptyset}{\gamma, \mathtt{empty}\ T \curvearrowright \mathtt{true}} \qquad \frac{\gamma, T \curvearrowright v \quad v \neq \emptyset}{\gamma, \mathtt{empty}\ T \curvearrowright \mathtt{false}}$$

$$\frac{\gamma, T \curvearrowright v \quad \gamma, x \in v, \Theta \curvearrowright^* v'}{\gamma, \bigcup\{e \mid x \in T\} \triangleright \Theta \curvearrowright v'}$$

$\boxed{\gamma, x \in v, \Theta \curvearrowright^* v'}$

$$\frac{}{\gamma, x \in \emptyset, \Theta \curvearrowright^* \emptyset} \qquad \frac{\gamma, x \in v_1, \Theta \curvearrowright^* v_1' \quad \gamma, x \in v_2, \Theta \curvearrowright^* v_2'}{\gamma, x \in v_1 \uplus v_2, \Theta \curvearrowright^* v_1' \uplus v_2'}$$

$$\frac{\ell_i \in \mathrm{dom}(\Theta) \quad \gamma[x \mapsto v_i], \Theta(\ell_i) \curvearrowright v_i'}{\gamma, x \in \{\ell_i.v_i\}, \Theta \curvearrowright^* \ell_i \cdot v_i'}$$

Figure 3. Trace replay.

values or delete set elements, but changes leading to additions of new labels to sets involved in comprehensions typically cannot be replayed.

Returning to the running example, suppose we change field B of row r_1 of R from 2 to 5. This change has no effect on the control flow choice taken in Q, and replaying the trace T succeeds. Likewise, changing the A or C field of any column of R has no effect, since these values do not affect the control flow choices in T. However, changing the B field of a row to 3 (or changing it from 3 to something else) means that replay will fail.

2.2 Key properties

Before moving on to consider trace slicing, we identify some properties that formalize the intuition that traces are consistent with and faithfully record execution.

Evaluation and traced evaluation are deterministic:

Proposition 2.3 (Determinacy)**.** *If $\gamma, e \Downarrow v, T$ and $\gamma, e \Downarrow v', T'$ then $v = v'$ and $T = T'$. If $\gamma, T \curvearrowright v$ and $\gamma, T \curvearrowright v'$ then $v = v'$.*

When the trace is irrelevant, we write $\gamma, e \Downarrow v$ to indicate that $\gamma, e \Downarrow v, T$ for some T.

Traces can be represented using pointers to share common subexpressions; using this DAG representation, traces can be stored in space polynomial in the input. (This sharing happens automatically in our implementation in Haskell.)

Proposition 2.4. *For a fixed e, if $\gamma, e \Downarrow v, T$ then the sizes of v and of the DAG representation of T are at most polynomial in $|\gamma|$.*

Proof. Most cases are straightforward. The only non-trivial case is for comprehensions, where we need a stronger induction hypothesis: if $\gamma, x \in v, e \Downarrow^* v', \Theta$ then the sizes of v' and of the DAG representation of Θ are at most polynomial in $|\gamma|$. □

Furthermore, traced evaluation produces a trace that replays to the same value as the original expression run on the original environment. We call this property *consistency*.

Proposition 2.5 (Consistency)**.** *If $\gamma, e \Downarrow v, T$ then $\gamma, T \curvearrowright v$.*

Finally, trace replay is faithful to ordinary evaluation in the following sense: if T is generated by running e in γ and we successfully replay T on γ' then we obtain the same value (and same trace) as if we had rerun e from scratch in γ', and vice versa:

Proposition 2.6 (Fidelity)**.** *If $\gamma, e \Downarrow v, T$, then for any γ', v' we have $\gamma', e \Downarrow v', T$ if and only if $\gamma', T \curvearrowright v'$.*

Observe that consistency is a special case of fidelity (with $\gamma' = \gamma, v' = v$. Moreover, the "if" direction of fidelity holds even though replay can fail, because we require that $\gamma', e \Downarrow v', T$ holds for the *same trace* T. If $\gamma', e \Downarrow v', T'$ is derivable but only for a different trace T', then replay fails.

3. Trace Slicing

The goal of the trace slicing algorithm we consider is to remove information from a trace and input that is not needed to recompute a part of the output. To accommodate these requirements, we introduce traces and values with *holes* and more generally, we consider *patterns* that represent relations on values capturing possible changes. In this section, we limit attention to pairs, and consider slicing for a class of *simple* patterns. We consider records and more expressive *enriched* patterns in the next section.

We extend traces with holes $\Box$ and define a subtrace relation $\sqsubseteq$ that is essentially a syntactic precongruence on traces and trace sets such that $\Box \sqsubseteq T$ holds and $\Theta \subseteq \Theta'$ implies $\Theta \sqsubseteq \Theta'$. (The definition of $\sqsubseteq$ is shown in full in the companion technical report.) Intuitively, holes denote parts of traces we do not care about, and we can think of a trace T with holes as standing for a set of possible complete traces $\{T' \mid T \sqsubseteq T'\}$ representing different ways of filling in the holes.

The syntax of *simple patterns* p, *set patterns* $\mathbf{sp}$, and *pattern environments* ρ is:

$$\begin{array}{rcl} p & ::= & \Box \mid \Diamond \mid \mathtt{c} \mid (p_1, p_2) \mid \mathbf{sp} \\ \mathbf{sp} & ::= & \emptyset \mid \{\ell_1.p_1, \ldots, \ell_n.p_n\} \\ \rho & ::= & [x_1 \mapsto p_1, \ldots, x_n \mapsto p_n] \end{array}$$

Essentially, a pattern is a value with holes in some positions. The meaning of each pattern is defined through a relation $\approx_p$ that says when two values are equivalent with respect to a pattern. This relation is defined in Figure 5. A hole $\Box$ indicates that the part of the value is unimportant, that is, for slicing purposes we don't care about that part of the result. Its associated relation $\approx_\Box$ relates any two values. An *identity pattern* $\Diamond$ is similar to a hole: it says that the value is important but its exact value is not specified, and its associated relation $\approx_\Diamond$ is the identity relation on values. Complete set patterns $\{\ell_1.p_1, \ldots, \ell_n.p_n\}$ specify the labels and patterns for all of the elements of a set; that is, such a pattern relates only sets that have exactly the labeled elements specified and whose corresponding values match according to the corresponding patterns.

We define the union of two set patterns as $\{\overrightarrow{\ell_i.p_i}\} \uplus \{\overrightarrow{\ell_i'.p_i'}\} = \{\overrightarrow{\ell_i.p_i}, \overrightarrow{\ell_i'.p_i'}\}$ provided their domains $\vec{\ell_i}$ and $\vec{\ell_i'}$ are prefix-disjoint. We define a (partial) least upper bound operation on patterns $p \sqcup p'$ such that for any v, v' we have $v \approx_{p \sqcup p'} v'$ if and only if $v \approx_p v'$ and $v \approx_{p'} v'$; the full definition is shown in Figure 4. We define

$$
\begin{array}{rcl}
\Box \sqcup p = p \sqcup \Box & = & p \\
\Diamond \sqcup p = p \sqcup \Diamond & = & p[\Diamond/\Box] \\
\mathsf{c} \sqcup \mathsf{c} & = & \mathsf{c} \\
(p_1, p_2) \sqcup (p_1', p_2') & = & (p_1 \sqcup p_1', p_2 \sqcup p_2') \\
\{\overline{\ell_i.p_i}\} \sqcup \{\overline{\ell_i.p_i'}\} & = & \{\overline{\ell_i.p_i \sqcup p_i'}\}
\end{array}
$$

where:

$$
\begin{array}{rcl}
\Diamond[\Diamond/\Box] = \Box[\Diamond/\Box] & = & \Diamond \\
\mathsf{c}[\Diamond/\Box] & = & \mathsf{c} \\
(p_1, p_2)[\Diamond/\Box] & = & (p_1[\Diamond/\Box], p_2[\Diamond/\Box]) \\
\{\overline{\ell_i.p_i}\}[\Diamond/\Box] & = & \{\overline{\ell_i.p_i[\Diamond/\Box]}\}
\end{array}
$$

Figure 4. Least upper bound for simple patterns

$$\boxed{v \approx_p v'}$$

$$\frac{}{v \approx_\Box v'} \qquad \frac{}{v \approx_\Diamond v} \qquad \frac{}{\mathsf{c} \approx_\mathsf{c} \mathsf{c}} \qquad \frac{v_1 \approx_{p_1} v_1' \qquad v_1 \approx_{p_2} v_2'}{(v_1, v_2) \approx_{(p_1,p_2)} (v_1', v_2')}$$

$$\frac{v_i \approx_{p_i} v_i' \quad (i \in \{1, \ldots, n\})}{\{\ell_1.v_1, \ldots, \ell_n.v_n\} \approx_{\{\ell_1.p_1, \ldots, \ell_n.p_n\}} \{\ell_1.v_1', \ldots, \ell_n.v_n'\}}$$

$$\gamma \approx_\rho \gamma' \iff \forall x \in dom(\rho).\ \gamma(x) \approx_{\rho(x)} \gamma'(x)$$

Figure 5. Simple pattern equivalence.

the partial ordering $p \sqsubseteq p'$ as $p \sqcup p' = p'$. We say that a value v *matches* pattern p if $p \sqsubseteq v$. Observe that this implies $v \approx_p v$. We extend the $\sqcup$ and $\sqsubseteq$ operations to pattern environments ρ pointwise, that is, $(\rho \sqcup \rho')(x) = \rho(x) \sqcup \rho(x')$.

We define several additional operations on patterns that are needed for the slicing algorithm. Consider the following *singleton extraction* operation $p.\epsilon$ and *label projection* operation $p[\ell]$:

$$
\begin{array}{rcll}
(\{\epsilon.p\}).\epsilon & = & p & \Box.\epsilon = \Box \quad \Diamond.\epsilon = \Diamond \\
\mathbf{sp}[\ell] & = & \{\ell'.v \mid \ell.\ell'.v \in \mathbf{sp}\} & \Box[\ell] = \Box \quad \Diamond[\ell] = \Diamond
\end{array}
$$

These operations are only used for the above cases; they have no effect on constant or pair patterns. For sets, $p[\ell]$ extracts the subset of p whose labels start with ℓ, truncating the initial prefix ℓ, while if p is $\Box$ or $\Diamond$ then again $p[\ell]$ returns the same kind of hole. Moreover, $\mathrm{dom}(p[\ell])$ is a prefix code if $\mathrm{dom}(p)$ is.

Suppose L is a prefix code. We define *restriction* of a set pattern p to L as follows:

$$\mathbf{sp}|_L = \{\ell.\ell'.p \in \mathbf{sp} \mid \ell \in L\} \qquad \Box|_L = \Box \qquad \Diamond|_L = \Diamond$$

It is easy to see that $\mathrm{dom}(p|_L) \subseteq \mathrm{dom}(p)$ so $\mathrm{dom}(p|_L)$ is a prefix code if $\mathrm{dom}(p)$ is, so this operation is well-defined on collections:

Lemma 3.1. *If* $\mathbf{sp}$ *is prefix-labeled and* L *is a prefix code then* $\mathbf{sp}[\ell]$ *and* $\mathbf{sp}|_L$ *are prefix-labeled.*

Backward Slicing The rules for backward slicing are given in Figure 6. The judgment $p, T \searrow \rho, T'$ slices trace T with respect to a pattern p to yield the slice T' and sliced input environment ρ. The sliced input environment records what parts of the input are needed to produce p; this is needed for slicing operations such as let-binding or comprehensions. The main new ideas are in the rules for collection operations, particularly comprehensions. The slicing rules SCONST, SPRIM, SVAR, SLET, SPAIR, SPROJ$_i$, and SIF follow essentially the same idea as in our previous work [1, 28]. We focus discussion on the new cases, but we review the key ideas for these operations here in order to make the presentation self-contained.

The rules for collections use labels and set pattern operations to effectively undo the evaluation of the set pattern. The rule SEMPTY is essentially the same as the constant rule. The rule SSNG uses the singleton extraction operation $p.\epsilon$ to obtain a pattern describing the single element of a singleton set value matching p. The rule SUNION uses the two projections $p[1]$ and $p[2]$ to obtain the patterns describing the subsets obtained from the first and second subtraces in the union pattern, respectively.

The rule SCOMP uses a similar idea to let-binding. We slice the trace set Θ using an auxiliary judgment $p, x.\Theta \searrow^* \rho, \Theta_0, p_0$, obtaining a sliced input environment, sliced trace set Θ_0, and pattern p_0 describing the set of values to which x was bound. We then use p_0 to slice backwards through the subtrace T that constructed the set. The auxiliary slicing judgment for trace sets has three rules: a trivial rule SEMPTY* when the set is empty, rule SSNG* that uses label projection $p[\ell]$ to handle a singleton trace set, and a rule SUNION* handling larger trace sets by decomposing them into subsets. This is essentially a structural recursion over the trace set, and is deterministic even though the rules can be used to decompose the trace sets in many different ways, because $\uplus$ is associative. Rule SUNION* also requires that we restrict the set pattern to match the domains of the corresponding trace patterns.

The slicing rules SSUM and SEMPTYP follow the same idea as for primitive operations at base type: we require that the whole set value be preserved exactly. As discussed by Perera et al. [28] with primitive operations, this is a potential source of overapproximation, since (for example) for an emptiness test, all we really need is to preserve the number of elements in the set, not their values. The last rule shows how to slice pair patterns when the pattern is $\Diamond$: we slice both of the subtraces by $\Diamond$ and combine the results.

It is straightforward to show that the slicing algorithm is well-defined for consistent traces. That is, if $\gamma, T \curvearrowright v$ and $p \sqsubseteq v$ then there exists ρ, S such that $p, T \searrow \rho, S$ holds, where $\rho \sqsubseteq \gamma$ and $S \sqsubseteq T$. We defer the correctness theorem for slicing using simple patterns to the end of the next section, since it is a special case of correctness for slicing using enriched patterns.

Continuing our running example, consider the pattern $p = \{[r_2].\langle A{:}\Box, B{:}8\rangle, [r_3].\Box\}$. The slice of T with respect to this pattern is of the form:

$$
T' = \bigcup\{\_ \mid x \in R\} \triangleright \{ \quad \begin{array}{l} [r_1].\Box, \\ [r_2].\mathtt{if}(x.B = 3, \_, \_) \triangleright_{\mathtt{true}} \{\_\}, \\ [r_3].\Box \ \} \end{array}
$$

and the slice of R is $R' = \{[r_1].\Box, [r_2].\langle A{:}\Box, B{:}3, C{:}8\rangle, [r_3].\Box\}$. (That is, R' is the value of $\rho(R)$, where ρ is the pattern environment produced by slicing T with respect to p.) Observe that the value of A is not needed but the value of B must remain 3 in order to preserve the control flow behavior of the trace on r_2. The holes indicate that changes to r_1 and r_3 in the input cannot affect r_2. However, a trace matching T' cannot be replayed if any of r_1, r_2, r_3 are deleted from the input or if the A field is removed from an input record, because the replay rules require all of the labels mentioned in collections or records in T' to be present. We now turn our attention to enriched patterns, which mitigate these drawbacks.

4. Enriched Patterns

So far we have considered only complete set patterns of the form $\{\ell_1.p_1, \ldots, \ell_n.p_n\}$. These patterns relate pairs of values that have exactly n elements labeled $\ell_1, \ldots, \ell_n$, each of which matches $p_1, \ldots, p_n$ respectively.

This is awkward, as we already can observe in our running example above: to obtain a slice describing how one record was computed, we need to use a set pattern that lists all of the indexes in the output. Moreover, the slice with respect to such a pattern may also include labeled subtraces explaining why the other elements exist in the output (and no others). This information seems intuitively

$\boxed{p, T \searrow \rho, S}$

$$\frac{}{\Box, T \searrow [], \Box}\ \textsc{SHole} \qquad \frac{}{p, \mathtt{c} \searrow [], \mathtt{c}}\ \textsc{SConst} \qquad \frac{\diamond, T_1 \searrow \rho_1, S_1 \quad \cdots \quad \diamond, T_n \searrow \rho_n, S_n}{p, \mathtt{f}(T_1, \ldots, T_n) \searrow \rho_1 \sqcup \cdots \sqcup \rho_n, \mathtt{f}(S_1, \ldots, S_n)}\ \textsc{SPrim} \qquad \frac{}{p, x \searrow [x \mapsto p], x}\ \textsc{SVar}$$

$$\frac{p_2, T_2 \searrow \rho_2[x \mapsto p_1], S_2 \qquad p_1, T_1 \searrow \rho_1, S_1}{p_2, \mathtt{let}\ x = T_1\ \mathtt{in}\ T_2 \searrow \rho_1 \sqcup \rho_2, \mathtt{let}\ x = S_1\ \mathtt{in}\ S_2}\ \textsc{SLet} \qquad \frac{p_1, T_1 \searrow \rho_1, S_1 \qquad p_2, T_2 \searrow \rho_2, S_2}{(p_1, p_2), (T_1, T_2) \searrow \rho_1 \sqcup \rho_2, (S_1, S_2)}\ \textsc{SPair} \qquad \frac{(p, \Box), T \searrow \rho, S}{p, T.\#_1 \searrow \rho, S.\#_1}\ \textsc{SProj}_1$$

$$\frac{(\Box, p), T \searrow \rho, S}{p, T.\#_2 \searrow \rho, S.\#_2}\ \textsc{SProj}_2 \qquad \frac{p, T' \searrow \rho', S' \qquad b, T \searrow \rho, S}{p, \mathtt{if}(T, e_1, e_2) \triangleright_b T' \searrow \rho' \sqcup \rho, \mathtt{if}(S, e_1, e_2) \triangleright_b S'}\ \textsc{SIf} \qquad \frac{}{p, \emptyset \searrow [], \emptyset}\ \textsc{SEmpty} \qquad \frac{p.\epsilon, T \searrow \rho, S}{p, \{T\} \searrow \rho, \{S\}}\ \textsc{SSng}$$

$$\frac{p[1], T_1 \searrow \rho_1, S_1 \qquad p[2], T_2 \searrow \rho_2, S_2}{p, T_1 \ \cup\ T_2 \searrow \rho_1 \sqcup \rho_2, S_1 \ \cup\ S_2}\ \textsc{SUnion} \qquad \frac{p, x.\Theta \searrow^* \rho', \Theta', p' \qquad p', T \searrow \rho, S}{p, \bigcup\{e |\ x \in T\} \triangleright \Theta \searrow \rho \sqcup \rho', \bigcup\{e |\ x \in S\} \triangleright \Theta'}\ \textsc{SComp}$$

$$\frac{\diamond, T \searrow \rho, S}{p, \mathtt{sum}\ T \searrow \rho, \mathtt{sum}\ S}\ \textsc{SEmptyP} \qquad \frac{\diamond, T \searrow \rho, S}{p, \mathtt{empty}\ T \searrow \rho, \mathtt{empty}\ S}\ \textsc{SSum} \qquad \frac{\diamond, T_1 \searrow \rho_1, S_1 \qquad \diamond, T_2 \searrow \rho_2, S_2}{\diamond, (T_1, T_2) \searrow \rho_1 \sqcup \rho_2, (S_1, S_2)}\ \textsc{SDiamond}$$

$\boxed{p, x.\Theta \searrow^* \rho, \Theta_0, p_0}$

$$\frac{}{\emptyset, x.\emptyset \searrow^* [], \emptyset, \emptyset}\ \textsc{SEmpty}^* \qquad \frac{p[\ell], T \searrow \rho[x \mapsto p_0], S}{p, x.\{\ell.T\} \searrow^* \rho, \{\ell.S\}, \{\ell.p_0\}}\ \textsc{SSng}^*$$

$$\frac{p|_{\mathrm{dom}(\Theta_1)}, x.\Theta_1 \searrow^* \rho_1, \Theta_1', p_1 \qquad p|_{\mathrm{dom}(\Theta_2)}, x.\Theta_2 \searrow^* \rho_2, \Theta_2', p_2}{p, x.\Theta_1 \uplus \Theta_2 \searrow^* \rho_1 \sqcup \rho_2, \Theta_1' \uplus \Theta_2', p_1 \uplus p_2}\ \textsc{SUnion}^*$$

Figure 6. Backward trace slicing.

irrelevant to the value at ℓ_1, and can be a major overhead if the collection is large. We also have considered only binary pairs; it would be more convenient to support record patterns directly.

In this section we sketch how to enrich the language of patterns to allow for *partial set* and *partial record patterns*, as follows:

$$\begin{array}{rcl} p & ::= & \Box \mid \diamond \mid \mathtt{c} \mid \mathbf{sp} \mid \mathbf{rp} \\ \mathbf{rp} & ::= & \langle\rangle \mid \langle \overline{A_i : p_i} \rangle \mid \langle \overline{A_i : p_i}; \Box \rangle \mid \langle \overline{A_i : p_i}; \diamond \rangle \\ \mathbf{sp} & ::= & \emptyset \mid \{\overline{\ell_i.p_i}\} \mid \{\overline{\ell_i.p_i}\} \dot\cup \Box \mid \{\overline{\ell_i.p_i}\} \dot\cup \diamond \end{array}$$

Record patterns are of the form $\langle \overline{A_i : p_i} \rangle$, listing the fields and the patterns they must match, possibly followed by $\Box$ or $\diamond$, which the remainder of the record must match. The pattern $\{\overline{\ell_i.p_i}\} \dot\cup \Box$ stands for a set with labeled elements matching patterns $p_1, \ldots, p_n$, plus some additional elements whose values we don't care about. For example, we can use the pattern $\{\ell_1.p_1\} \dot\cup \Box$ to express interest in why element ℓ_1 matches p_1, when we don't care about the rest of the set. The second partial pattern, $\{\overline{\ell_i.p_i}\} \dot\cup \diamond$, has similar behavior, but it says that the rest of the sets being considered must be equal. For example, $\{\ell.\Box\} \dot\cup \diamond$ says that the two sets are equal except possibly at label ℓ. This pattern is needed mainly in order to ensure that we can define a least upper bound on enriched patterns, since we cannot express $(\{\ell_1.v_1, \ldots, \ell_n.p_n\} \dot\cup \Box) \sqcup \diamond$ otherwise.

We define $\mathrm{dom}(\{\overline{\ell_i.p_i}\} \dot\cup \Box) = \mathrm{dom}(\{\overline{\ell_i.p_i}\} \dot\cup \diamond) = \{\ell_1, \ldots, \ell_n\}$. Disjoint union of enriched patterns $\mathbf{sp} \uplus \mathbf{sp}'$ is defined only if the domains are prefix-disjoint, so that the labels of the result still form a prefix code; this operation is defined in Figure 7.

We now extend the definitions of $p[\diamond/\Box]$ and $\sqcup$ to account for extended patterns. We extend the $[\diamond/\Box]$ substitution operation as follows:

$$\begin{array}{rcl} \langle \overline{A_i : p_i} \rangle[\diamond/\Box] & = & \langle \overline{A_i : p_i[\diamond/\Box]} \rangle \\ \langle \overline{A_i : p_i}; \Box \rangle[\diamond/\Box] & = & \langle \overline{A_i : p_i[\diamond/\Box]}; \diamond \rangle \\ \langle \overline{A_i : p_i}; \diamond \rangle[\diamond/\Box] & = & \langle \overline{A_i : p_i[\diamond/\Box]}; \diamond \rangle \\ (\{\overline{\ell_i.p_i}\} \dot\cup \Box)[\diamond/\Box] & = & \{\ell_i.p_i[\diamond/\Box]\} \dot\cup \diamond \\ (\{\overline{\ell_i.p_i}\} \dot\cup \diamond)[\diamond/\Box] & = & \{\ell_i.p_i[\diamond/\Box]\} \dot\cup \diamond \end{array}$$

We handle the additional cases of the $\sqcup$ operation in Figure 8, and extend the $\approx_p$ relation as shown in Figure 9. Note that taking the least upper bound of a partial pattern with a complete pattern yields a complete pattern, while taking the least upper bound of

$$\begin{array}{rcl} \diamond \uplus \Box = \Box \uplus \diamond = \Box \uplus \Box & = & \Box \\ \diamond \uplus \diamond & = & \diamond \\ \mathbf{sp} \uplus \emptyset = \emptyset \uplus \mathbf{sp} & = & \mathbf{sp} \\ \{\overline{\ell_i.p_i}\} \uplus \Box = \Box \uplus \{\overline{\ell_i.p_i}\} & = & \{\overline{\ell_i.p_i}\} \dot\cup \Box \\ \{\overline{\ell_i.p_i}\} \uplus \diamond = \diamond \uplus \{\overline{\ell_i.p_i}\} & = & \{\overline{\ell_i.p_i}\} \dot\cup \diamond \\ \{\overline{\ell_i.p_i}\} \uplus (\{\overline{\ell'_j.p'_j}\} \dot\cup \Box) & = & (\{\overline{\ell_i.p_i}\} \uplus \{\overline{\ell'_j.p'_j}\}) \dot\cup \Box \\ \{\overline{\ell_i.p_i}\} \uplus (\{\overline{\ell'_j.p'_j}\} \dot\cup \diamond) & = & (\{\overline{\ell_i.p_i}\} \uplus \{\overline{\ell'_j.p'_j}\}) \dot\cup \diamond \\ (\{\overline{\ell_i.p_i}\} \dot\cup \Box) \uplus (\{\overline{\ell'_j.p'_j}\} \dot\cup \Box) & = & (\{\overline{\ell_i.p_i}\} \uplus \{\overline{\ell'_j.p'_j}\}) \dot\cup \Box \\ (\{\overline{\ell_i.p_i}\} \dot\cup \diamond) \uplus (\{\overline{\ell'_j.p'_j}\} \dot\cup \Box) & = & (\{\overline{\ell_i.p_i}\} \uplus \{\overline{\ell'_j.p'_j}\}) \dot\cup \Box \\ (\{\overline{\ell_i.p_i}\} \dot\cup \Box) \uplus (\{\overline{\ell'_j.p'_j}\} \dot\cup \diamond) & = & (\{\overline{\ell_i.p_i}\} \uplus \{\overline{\ell'_j.p'_j}\}) \dot\cup \Box \\ (\{\overline{\ell_i.p_i}\} \dot\cup \diamond) \uplus (\{\overline{\ell'_j.p'_j}\} \dot\cup \diamond) & = & (\{\overline{\ell_i.p_i}\} \uplus \{\overline{\ell'_j.p'_j}\}) \dot\cup \diamond \end{array}$$

Figure 7. Enriched pattern union

partial patterns involving $\diamond$ again relies on the $[\diamond/\Box]$ substitution operation.

We extend the singleton extraction $p.\epsilon$, label projection $p[\ell]$, and restriction $p|_L$ operations on set patterns as follows:

$$\begin{array}{rcl} (\{\epsilon.p\} \dot\cup \diamond).\epsilon = (\{\epsilon.p\} \dot\cup \Box).\epsilon & = & p \\ \ell \cdot (\{\overline{\ell_i.p_i}\} \dot\cup \Box) & = & (\ell \cdot \{\overline{\ell_i.p_i}\}) \uplus \Box \\ \ell \cdot (\{\overline{\ell_i.p_i}\} \dot\cup \diamond) & = & (\ell \cdot \{\overline{\ell_i.p_i}\}) \uplus \diamond \\ (\{\overline{\ell_i.p_i}\} \dot\cup \Box)[\ell] & = & (\{\overline{\ell_i.p_i}\}[\ell]) \uplus \Box \\ (\{\overline{\ell_i.p_i}\} \dot\cup \diamond)[\ell] & = & (\{\overline{\ell_i.p_i}\}[\ell]) \uplus \diamond \\ (\{\overline{\ell_i.p_i}\} \dot\cup \Box)|_L & = & (\{\overline{\ell_i.p_i}\}|_L) \uplus \Box \\ (\{\overline{\ell_i.p_i}\} \dot\cup \diamond)|_L & = & (\{\overline{\ell_i.p_i}\}|_L) \uplus \diamond \end{array}$$

Note that in many cases, we use the disjoint union operation $\uplus$ on the right-hand side; this ensures, for example, that we never produce results of the form $\emptyset \dot\cup \Box$ or $\emptyset \dot\cup \diamond$; these are normalized to $\Box$ and $\diamond$ respectively, and this normalization reduces the number of corner cases in the slicing algorithm.

We define a record pattern projection operation $p.A$ as follows:

$$\begin{array}{rcll} \langle A_1{:}p_1, \ldots, A_n{:}p_n \rangle.A_i & = & p_i & \Box.A = \Box \\ \langle A_1{:}p_1, \ldots, A_n{:}p_n; \_ \rangle.A_i & = & p_i & \diamond.A = \diamond \\ \langle A_1{:}p_1, \ldots, A_n{:}p_n; \Box \rangle.B & = & \Box & (B \notin \{A_1, \ldots, A_n\}) \\ \langle A_1{:}p_1, \ldots, A_n{:}p_n; \diamond \rangle.B & = & \diamond & (B \notin \{A_1, \ldots, A_n\}) \end{array}$$

We extend the slicing judgment to accommodate these new patterns in Figure 10. The rules $\textsc{SRec}$ and $\textsc{SProj}_A$ are similar to

$$
\begin{array}{rcl}
\{\overline{\ell_i.p_i}, \overline{\ell'_j.q_j}\} \sqcup (\{\overline{\ell_i.p'_i}\} \dot\cup \Box) &=& \{\overline{\ell_i.p_i \sqcup p'_i}, \overline{\ell'_j.q_j}\} \\
\{\overline{\ell_i.p_i}, \overline{\ell'_j.q_j}\} \sqcup (\{\overline{\ell_i.p'_i}\} \dot\cup \Diamond) &=& \{\overline{\ell_i.p_i \sqcup p'_i}, \overline{\ell'_j.q_j[\Diamond/\Box]}\} \\
(\{\overline{\ell_i.p_i}, \overline{\ell'_j.q_j}\} \dot\cup \Box) \sqcup (\{\overline{\ell_i.p'_i}, \overline{\ell''_k.r_k}\} \dot\cup \Box) &=& \{\overline{\ell_i.p_i \sqcup p'_i}, \overline{\ell'_j.q_j}, \overline{\ell''_k.r_k}\} \dot\cup \Box \\
(\{\overline{\ell_i.p_i}, \overline{\ell'_j.q_j}\} \dot\cup \Box) \sqcup (\{\overline{\ell_i.p'_i}, \overline{\ell''_k.r_k}\} \dot\cup \Diamond) &=& \{\overline{\ell_i.p_i \sqcup p'_i}, \overline{\ell'_j.q_j[\Diamond/\Box]}, \overline{\ell''_k.r_k}\} \dot\cup \Diamond \\
(\{\overline{\ell_i.p_i}, \overline{\ell'_j.q_j}\} \dot\cup \Diamond) \sqcup (\{\overline{\ell_i.p'_i}, \overline{\ell''_k.r_k}\} \dot\cup \Diamond) &=& \{\overline{\ell_i.p_i \sqcup p'_i}, \overline{\ell'_j.q_j[\Diamond/\Box]}, \overline{\ell''_k.r_k[\Diamond/\Box]}\} \dot\cup \Diamond \\
\\
\langle \overline{A_i : p_i}, \overline{B_j : q_j} \rangle \sqcup (\langle \overline{A_i : p'_i}; \Box \rangle) &=& \langle \overline{A_i : p_i \sqcup p'_i}, \overline{B_j : q_j} \rangle \\
\langle \overline{A_i : p_i}, \overline{B_j : q_j} \rangle \sqcup (\langle \overline{A_i : p'_i}; \Diamond \rangle) &=& \langle \overline{A_i : p_i \sqcup p'_i}, \overline{B_j : q_j[\Diamond/\Box]} \rangle \\
(\langle \overline{A_i : p_i}, \overline{B_j : q_j}; \Box \rangle) \sqcup (\langle \overline{A_i : p'_i}, \overline{C_k : r_k}; \Box \rangle) &=& \langle \overline{A_i : p_i \sqcup p'_i}, \overline{B_j : q_j}, \overline{C_k : r_k}; \Box \rangle \\
(\langle \overline{A_i : p_i}, \overline{B_j : q_j}; \Box \rangle) \sqcup (\langle \overline{A_i : p'_i}, \overline{C_k : r_k}; \Diamond \rangle) &=& \langle \overline{A_i : p_i \sqcup p'_i}, \overline{B_j : q_j[\Diamond/\Box]}, \overline{C_k : r_k}; \Diamond \rangle \\
(\langle \overline{A_i : p_i}, \overline{B_j : q_j}; \Diamond \rangle) \sqcup (\langle \overline{A_i : p'_i}, \overline{C_k : r_k}; \Diamond \rangle) &=& \langle \overline{A_i : p_i \sqcup p'_i}, \overline{B_j : q_j[\Diamond/\Box]}, \overline{C_k : r_k[\Diamond/\Box]}; \Diamond \rangle
\end{array}
$$

Figure 8. Least upper bound for enriched patterns (excluding some symmetric cases).

$$\frac{v_1 \approx_{p_1} v'_1 \quad \cdots \quad v_n \approx_{p_n} v'_n}{\langle \overline{A_i : v_i} \rangle \approx_{\langle \overline{A_i : p_i} \rangle} \langle \overline{A_i : v'_i} \rangle}$$

$$\frac{v_1 \approx_{p_1} v'_1 \quad \cdots \quad v_n \approx_{p_n} v'_n}{\langle \overline{A_i : v_i}, \overline{B_j : w_j} \rangle \approx_{\langle \overline{A_i : p_i}; \Box \rangle} \langle \overline{A_i : v'_i}, \overline{C_k : w'_k} \rangle}$$

$$\frac{v_1 \approx_{p_1} v'_1 \quad \cdots \quad v_n \approx_{p_n} v'_n}{\langle \overline{A_i : v_i}, \overline{B_j : w_j} \rangle \approx_{\langle \overline{A_i : p_i}; \Diamond \rangle} \langle \overline{A_i : v'_i}, \overline{B_j : w_j} \rangle}$$

$$\frac{v_1 \approx_{p_1} v'_1 \quad \cdots \quad v_n \approx_{p_n} v'_n}{\{\overline{\ell_i.v_i}\} \uplus v \approx_{\{\overline{\ell_i.p_i}\} \dot\cup \Box} \{\overline{\ell_i.v'_i}\} \uplus v'}$$

$$\frac{v_1 \approx_{p_1} v'_1 \quad \cdots \quad v_n \approx_{p_n} v'_n}{\{\overline{\ell_i.v_i}\} \uplus v \approx_{\{\overline{\ell_i.p_i}\} \dot\cup \Diamond} \{\overline{\ell_i.v'_i}\} \uplus v}$$

Figure 9. Enriched pattern equivalence

$$\boxed{p, T \searrow \rho, S} \qquad \frac{\langle A{:}p; \Box \rangle, T \searrow \rho, S}{p, T.A \searrow \rho, S.A}\ \textsc{SRec}$$

$$\frac{p.A_1, T_1 \searrow \rho_1, S_1 \quad \cdots \quad p.A_n, T_n \searrow \rho_n, S_n}{p, \langle A_1{:}T_1, \ldots, A_n{:}T_n \rangle \searrow \rho_1 \sqcup \cdots \sqcup \rho_n, \langle A_1{:}S_1, \ldots, A_n{:}S_n \rangle}\ \textsc{SProj}_A$$

$$\boxed{p, x.\Theta \searrow^* \rho, \Theta', p_0}$$

$$\frac{}{\Box, x.\Theta \searrow^* [], \emptyset, \Box}\ \textsc{SHole}^* \qquad \frac{}{\Diamond, x.\emptyset \searrow^* [], \emptyset, \emptyset}\ \textsc{SDiamond}^*$$

Figure 10. Backward trace slicing over enriched patterns.

those for pairs, except that we use the field projection operation in the case for a record trace, and we use partial record patterns $\langle A : p; \Box \rangle$ in the case for a field projection trace. The added rules SHOLE* and SDIAMOND* handle the possibility that a partial pattern reduces to $\Box$ or $\Diamond$ through projection; we did not need to handle this case earlier because a simple set pattern is either a hole (which could be handled by the rule $\Box, T \searrow [], \Box$) or a complete set pattern showing all of the labels of the result.

Both simple and extended patterns satisfy a number of lemmas that are required to prove the correctness of trace slicing.

Lemma 4.1 (Properties of union and restriction)**.**

1. *If $p_1 \sqsubseteq v_1$ and $p_2 \sqsubseteq v_2$ and $v_1 \approx_{p_1} v'_1$ and $v_2 \approx_{p_2} v'_2$ then $v_1 \uplus v_2 \approx_{p_1 \uplus p_2} v'_1 \uplus v'_2$, provided all of these disjoint unions are defined.*
2. *If $p \sqsubseteq v_1 \uplus v_2$ and $L_1 \leq \mathrm{dom}(v_1)$ and $L_2 \leq \mathrm{dom}(v_2)$ and L_1, L_2 are prefix-disjoint, then $p|_{L_1} \sqsubseteq v_1$ and $p|_{L_2} \sqsubseteq v_2$.*

Lemma 4.2 (Projection and $\sqsubseteq$)**.**

1. *If $p \sqsubseteq \{\epsilon.v\}$ then $p.\epsilon \sqsubseteq v$.*
2. *If $p \sqsubseteq 1 \cdot v_1 \uplus 2 \cdot v_2$ then $p[1] \sqsubseteq v_1$ and $p[2] \sqsubseteq v_2$.*
3. *If $p \sqsubseteq \ell \cdot v$ then $p[\ell] \sqsubseteq v$.*
4. *If $p \sqsubseteq \langle \overline{A_i : v_i} \rangle$ then $p.A_i \sqsubseteq v_i$.*

Lemma 4.3 (Projection and $\approx_p$)**.**

1. *If $p \sqsubseteq \{\epsilon.v\}$ and $v \approx_{p.\epsilon} v'$ then $\{\epsilon.v\} \approx_p \{\epsilon.v'\}$.*
2. *If $p \sqsubseteq 1 \cdot v_1 \uplus 2 \cdot v_2$ and $v_1 \approx_{p[1]} v'_1$ and $v_2 \approx_{p[2]} v'_2$ then $1 \cdot v_1 \uplus 2 \cdot v_2 \approx_p 1 \cdot v'_1 \uplus 2 \cdot v'_2$.*
3. *If $p \sqsubseteq \ell \cdot v$ and $v \approx_{p[\ell]} v'$ then $\ell \cdot v \approx_p \ell \cdot v'$.*
4. *If $p \sqsubseteq \langle \overline{A_i : v_i} \rangle$ and $v_1 \approx_{p.A_1} v'_1, \ldots, v_n \approx_{p.A_n} v'_n$ then $\langle \overline{A_i : v_i} \rangle \approx_p \langle \overline{A_i : v'_i} \rangle$.*

We now state the key correctness property for slicing. Intuitively, it says that if we slice T with respect to output pattern p, obtaining a slice ρ and S, then p will be reproduced on recomputation under any change to the input and trace that is consistent with the slice — formally, that means that the changed trace T' must match the sliced trace S, and the changed input γ' must match γ modulo ρ.

Theorem 4.4 (Correctness of Slicing)**.** *Suppose $\gamma, T \curvearrowright v$ and $p \sqsubseteq v$ and $p, T \searrow \rho, S$. Then for all $\gamma' \approx_\rho \gamma$ and $T' \sqsupseteq S$ such that $\gamma', T' \curvearrowright v'$ we have $v' \approx_p v$.*

Returning to our running example, we can use the enriched pattern $p' = \{[r_2].\langle B{:}8; \Box \rangle\} \dot\cup \Box$ to indicate interest in the B field of r_2, without naming the other fields of the row or the other row indexes. Slicing with respect to this pattern yields the following slice:

$$T'' = \bigcup \{\_ \mid x \in R\} \triangleright \{[r_2].\texttt{if}(x.B = 3, \_, \_) \triangleright_{\texttt{true}} \{\_\}\}$$

and the slice of R is $R'' = \{[r_2].\langle B{:}3, C{:}8; \Box \rangle\} \dot\cup \Box$. This indicates that (as before) the values of r_1 and r_3 and of the A field of r_2 are irrelevant to r_2 in the result; unlike T', however, we can also potentially replay if r_1 and r_3 have been deleted from R. Likewise, we can replay if the A field has been removed from a record, or if some other field such as D is added. This illustrates that enriched

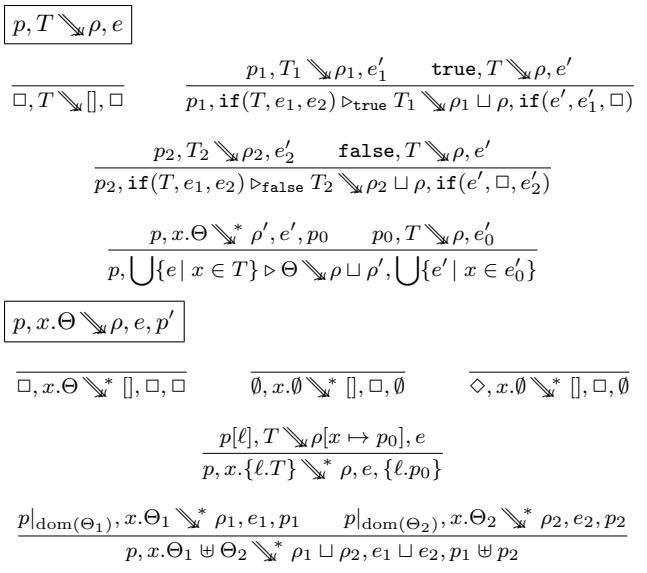

Figure 11. Unevaluation (selected rules).

patterns allow for smaller slices than simple patterns, with greater flexibility concerning possible updates to the input.

A natural question is whether slicing computes the (or a) smallest possible answer. Because our definition of correct slices is based on recomputation, minimal slices are not computable, by a straightforward reduction from the undecidability of minimizing dependency provenance [1, 12].

5. Query and Differential Slicing

We now adapt slicing techniques to provide explanations in terms of query expressions, and show how to use differences between slices to provide precise explanations for parts of the output.

5.1 Query slicing

Our previous work [28] gave an algorithm for extracting a *program slice* from a trace. We now adapt this idea to queries. A trace slice shows the parts of the trace that need to be replayed in order to compute the desired part of the output; similarly, a *query slice* shows the part of the query expression that is needed in order to ensure that the desired part of the output is recomputed. As with traces, we allow holes $\Box$ in programs to allow deleting subexpressions, and define $\sqsubseteq$ as a syntactic precongruence such that $\Box \sqsubseteq e$. We also define a least upper bound operation $e \sqcup e'$ on partial query expressions in the obvious way, so that $\Box \sqcup e = e$.

We define a judgment $p, T \searrow \rho, e$ that traverses T and "unevaluates" p, yielding a partial input environment ρ and partial program e. The rules are illustrated in Figure 11. Many of the rules are similar to those for trace slicing; the main differences arise in the cases for conditionals and comprehensions, where we collapse the sliced expressions back into expressions, possibly inserting holes or merging sliced expressions obtained by running the same code in different ways (as in a comprehension that contains a conditional).

Again, it is straightforward to show that if $\gamma, e \Downarrow v, T$ and $p \sqsubseteq v$ then there exist ρ, e' such that $p, T \searrow \rho, e'$, where $\rho \sqsubseteq \gamma$ and $e' \sqsubseteq e$. The essential correctness property for query slices is similar to that for trace slices: again, we require that rerunning any sufficiently similar query on a sufficiently similar input produces a result that matches p.

Theorem 5.1 (Correctness of Query Slicing)**.** *Suppose $\gamma, T \curvearrowright v$ and $p \sqsubseteq v$ and $p, T \searrow \rho, e$. Then for all $\gamma' \approx_\rho \gamma$ and $e' \sqsupseteq e$ such that $\gamma', e' \Downarrow v'$ we have $v' \approx_p v$.*

Combining with the consistency property (Proposition 2.5), we have:

Corollary 5.2. *Suppose $\gamma, e \Downarrow v, T$ and $p \sqsubseteq v$ and $p, T \searrow \rho, e'$. Then for all $\gamma' \approx_\rho \gamma$ and $e'' \sqsupseteq e'$ such that $\gamma', e'' \Downarrow v'$, we have $v \approx_p v'$.*

Continuing our running example, the query slice for the pattern p' considered above is

$$Q' = \bigcup\{\texttt{if}(x.B = 3, \{\langle A{:}\Box, B{:}x.C\rangle\}, \{\}) \mid x \in R\}$$

since only the computation of the A field in the output is irrelevant to p'.

5.2 Differential slicing

We consider a *pattern difference* to be a pair of patterns (p, p') where $p \sqsubseteq p'$. Intuitively, a pattern difference selects the part of a value present in the outer component p' and not in the inner component p. For example, the pattern difference $(\langle B{:}\Box; \Box\rangle, \langle B{:}8; \Box\rangle)$ selects the value 8 located in the B component of a record. We can also write this difference as $\langle B{:}\boxed{8}; \Box\rangle$, using $\boxed{?}$ to highlight the boundary between the inner and outer pattern. Trace and query pattern differences are defined analogously.

It is straightforward to show by induction that slicing is monotonic in both arguments:

Lemma 5.3 (Monotonicity)**.** *If $p \sqsubseteq p'$ and $T \sqsubseteq T'$ and $p', T' \searrow \rho', S'$ then there exist ρ, S such that $p, T \searrow \rho, S$ and $\rho \sqsubseteq \rho'$ and $S \sqsubseteq S'$. In addition, $p, S' \searrow \rho, S$.*

This implies that given a pattern difference and a trace, we can compute a trace difference using the following rule:

$$\frac{p_2, T \searrow \rho_2, S_2 \qquad p_1, S_2 \searrow \rho_1, S_1}{(p_1, p_2), T \searrow (\rho_1, \rho_2), (S_1, S_2)}$$

It follows from monotonicity that $\rho_1 \sqsubseteq \rho_2$ and $S_1 \sqsubseteq S_2$, thus, pattern differences yield trace differences. Furthermore, the second part of monotonicity implies that we can compute the smaller slice S_1 from the larger slice S_2, rather than re-traverse T. It is also possible to define a simultaneous differential slicing judgment, as an optimization to ensure we only traverse the trace once.

Query slicing is also monotone, so differential query slices can be obtained in exactly the same way. Revisiting our running example one last time, consider the differential pattern $\{[r_2].\langle B{:}\boxed{8}; \Box\rangle\} \dot\cup \Box$. The differential query slice for the pattern p' considered above is

$$Q'' = \bigcup\{\texttt{if}(x.B = 3, \{\langle A{:}\Box, B{:}\boxed{x.C}\rangle\}, \{\}) \mid x \in R\}$$

6. Examples and Discussion

In this section we present some more complex examples to illustrate key points.

Renaming Recall the swapping query from the introduction, written in NRC as

$$Q_1 = \bigcup\{\{\texttt{if}(x.A > x.B, \langle A{:}x.B, B{:}x.A\rangle, \langle x\rangle)\} \mid x \in R\}$$

This query illustrates a key difference between our approach and the how-provenance model of Green et al. [21]. As discussed in [14], renaming operations are ignored by how-provenance, so the how-provenance annotations of the results of Q_1 are the same as for a query that simply returns R. In other words, the choice to swap the fields when $A > B$ is not reflected in the how-provenance,

which shows that it is impossible to extract where-provenance (or traces) from how-provenance. Extracting where-provenance from traces appears straightforward, extending our previous work [1].

This example also illustrates how traces and slices can be used for partial recomputation. The slice for output pattern $\{[1, r_1].\langle B{:}2\rangle\} \dot\cup \square$, for example, will show that this record was produced because the A component of $\langle A{:}1, B{:}2, C{:}7\rangle$ at index $[r_1]$ in the input was less than or equal to the B component. Thus, we can replay after any change that preserves this ordering information.

Union Consider query

$$Q_2 = \bigcup\{\{\langle B{:}x.B\rangle\} \mid x \in R\} \cup \{\langle B{:}3\rangle\}$$

that projects the B fields of elements of R and adds another copy of $\langle B{:}3\rangle$ to the result. This yields

$$Q_2(R) = \{[1, r_1].\langle B{:}2\rangle, [1, r_2].\langle B{:}3\rangle, [1, r_3].\langle B{:}3\rangle, [2].\langle B{:}3\rangle\}$$

This illustrates that the indexes may not all have the same length, but still form a prefix code. If we slice with respect to $\{[1, r_2].\langle B{:}3\rangle\} \dot\cup \square$ then the query slice is:

$$Q_2' = \bigcup\{\{\langle B{:}x.B\rangle\} \mid x \in R\} \cup \square$$

and $R' = \{[r_2].\langle B{:}3; \square\rangle\} \dot\cup \square$ whereas if we slice with respect to $\{[2].\langle B{:}3\rangle\} \dot\cup \square$ then the query slice is $Q_2'' = \square \cup \{\langle B{:}3\rangle\}$ and $R'' = \square$, indicating that this part of the result has no dependence on the input.

A related point: one may wonder whether it makes sense to select a particular copy of $\langle B{:}3\rangle$ in the output, since in a conventional multiset, multiple copies of the same value are indistinguishable. We believe it is important to be able to distinguish different copies of a value, which may have different explanations. Treating n copies of a value as a single value with multiplicity n would obscure this distinction and force us to compute the slices of all of the copies even if only a single explanation is required. This is why we have chosen to work with indexed sets, rather than pure multisets.

Joins So far all examples have involved a single table R. Consider a simple join query

$$Q_3 = \{\langle A{:}x.A, B{:}y.C\rangle \mid x \in R, y \in S, x.B = y.B\}$$

and consider the following table S, and the result $Q_3(R, S)$.

$$S = \begin{array}{c|cc} id & B & C \\ \hline [s_1] & 2 & 4 \\ [s_2] & 3 & 4 \\ [s_3] & 4 & 5 \end{array} \qquad Q_3(R,S) = \begin{array}{c|cc} id & A & B \\ \hline [r_1, s_1] & 1 & 4 \\ [r_2, s_2] & 2 & 4 \\ [r_3, s_2] & 4 & 5 \end{array}$$

The full trace of this query execution is as follows:

$$\begin{array}{ll} T_3 = \bigcup\{\_ \mid x \in R\} \triangleright \{ & \\ \quad [r_1].\bigcup\{\_ \mid y \in S\} \triangleright \{ & [s_1].\mathtt{if}(x.B = y.B, \_, \_) \triangleright_{\mathtt{true}} \{\_\}, \\ & [s_2].\mathtt{if}(x.B = y.B, \_, \_) \triangleright_{\mathtt{false}} \{\}, \\ & [s_3].\mathtt{if}(x.B = y.B, \_, \_) \triangleright_{\mathtt{false}} \{\}\}, \\ \quad [r_2].\bigcup\{\_ \mid y \in S\} \triangleright \{ & [s_1].\mathtt{if}(x.B = y.B, \_, \_) \triangleright_{\mathtt{false}} \{\}, \\ & [s_2].\mathtt{if}(x.B = y.B, \_, \_) \triangleright_{\mathtt{true}} \{\_\}, \\ & [s_3].\mathtt{if}(x.B = y.B, \_, \_) \triangleright_{\mathtt{false}} \{\}\}, \\ \quad [r_3].\bigcup\{\_ \mid y \in S\} \triangleright \{ & [s_1].\mathtt{if}(x.B = y.B, \_, \_) \triangleright_{\mathtt{false}} \{\}, \\ & [s_2].\mathtt{if}(x.B = y.B, \_, \_) \triangleright_{\mathtt{true}} \{\_\}, \\ & [s_3].\mathtt{if}(x.B = y.B, \_, \_) \triangleright_{\mathtt{false}} \{\}\}\} \end{array}$$

Slicing with respect to $\{[r_1, s_1].\langle A{:}1; \square\rangle, [r_2, s_2].\langle B{:}4; \square\rangle\} \dot\cup \square$ yields trace slice

$$\begin{array}{ll} T_3' = \bigcup\{\_ \mid x \in R\} \triangleright \{ & \\ \quad [r_1].\bigcup\{\_ \mid y \in S\} \triangleright \{ & [s_1].\mathtt{if}(x.B = y.B, \_, \_) \triangleright_{\mathtt{true}} \{\_\}, \\ \quad [r_2].\bigcup\{\_ \mid y \in S\} \triangleright \{ & [s_2].\mathtt{if}(x.B = y.B, \_, \_) \triangleright_{\mathtt{true}} \{\_\}\} \end{array}$$

and input slice $R_3' = \{[r_1].\langle A{:}1, B{:}2; \square\rangle, [r_2].\langle B{:}3; \square\rangle\}$ and $S_3' = \{[s_1].\langle B{:}2; \square\rangle, [s_2].\langle B{:}3; C{:}4\rangle\}$.

Workflows NRC expressions can be used to represent workflows, if primitive operations are added representing the workflow steps [2, 22]. To illustrate query slicing and differential slicing for a workflow-style query, consider the following more complex query:

$$Q_4 = \{f(x, y) \mid x \in T, y \in T, z \in U, p(x, y), q(x, y, z)\}$$

where f computes some function of x and y and p and q are selection criteria. Here, we assume T and U are collections of data files and f, p, q are additional primitive operations on them. This query exercises most of the distinctive features of our approach; we can of course translate it to the NRC core calculus used in the rest of the paper. If $dom(T) = \{t_1, \ldots, t_{10}\}$ and $dom(U) = \{u_1, \ldots, u_{10}\}$ then we might obtain result $\{[t_3, t_4, u_5].v_1, [t_6, t_8, u_{10}].v_2\}$. If we focus on the value v_1 using the pattern $\{[t_3, t_4, u_5].v_1\} \dot\cup \square$, then the program slice we obtain is Q_4 itself, while the data slice might be $T' = \{[t_3].\diamond, [t_4].\diamond\} \dot\cup \square, U' = \{[u_5].\diamond\} \dot\cup \square$, indicating that if the values at t_3, t_4, u_5 are held fixed then the end result will still be v_1. The trace slice is similar, and shows that v_1 was computed by applying f with x bound to the value at t_3 in T, y bound to t_4, and z bound to u_5, and that $p(x, y)$ and $q(x, y, z)$ succeeded for these values.

If we consider a differential slice using pattern difference $\{[t_3, t_4, u_5].\boxed{v_1}\} \dot\cup \square$ then we obtain the following program difference:

$$Q_4^\delta = \bigcup\{\boxed{f(x, y)} \mid x \in T, y \in T, z \in U, p(x, y), q(x, y, z)\}$$

This shows that most of the query is needed to ensure that the result at $[t_3, t_4, u_5]$ is produced, but the subterm $f(x, y)$ is only needed to compute the value v_1. This can be viewed as a query-based explanation for this part of the result.

7. Implementation

To validate our design and experiment with larger examples, we extended our Haskell implementation Slicer of program slicing for functional programs [28] with the traces and slicing techniques presented in this paper. We call the resulting system NRCSlicer; it supports a free combination of NRC and general-purpose functional programming features. NRCSlicer interprets expressions in-memory without optimization. As reported previously for Slicer, we have experimented with several alternative tracing and slicing strategies, which use Haskell's lazy evaluation strategy in different ways. The alternatives we consider here are:

- *eager*: the trace is fully computed during evaluation.
- *lazy*: the value is computed eagerly, but the trace is computed lazily using Haskell's default lazy evaluation strategy.

To evaluate the effectiveness of enriched patterns, we measured the time needed for the eager and lazy techniques to trace and slice the workflow example Q_4 in the previous section. We considered a instantiation of the workflow where the data values are simply integers and with input tables $T, U = \{1, \ldots, 50\}$, and defined the operations $f(x, y)$ as $x * y$, $p(x, y)$ as $x < y$, and $q(x, y, z)$ as $x^2 + y^2 = z^2$. This is not a realistic workflow, and we expect that the time to evaluate the basic operations of a realistic workflow following this pattern would be much larger. However, the overheads of tracing and slicing do not depend on the execution time of primitive operations, so we can still draw some conclusions from this simplistic example.

The comprehension iterates over 50^3 = 125,000 triples, producing 20 results. We considered simple and enriched patterns selecting a single element of the result. We measured evaluation time, and the overhead of tracing, trace slicing, and query slicing. The experiments were conducted on a MacBook Pro with 2GB RAM and a 2.8GHz Intel Core Duo, using GHC version 7.4.

	eval	trace	slice	qslice
eager-simple	0.5	1.5	2.5	1.6
eager-enriched	0.5	1.5	<0.1	<0.1
lazy-simple	0.5	0.7	1.3	1.7
lazy-enriched	0.5	0.7	<0.1	<0.1

The times are in seconds. The "eval" column shows the time needed to compute the result without tracing. The "trace", "slice", and "qslice" columns show the added time needed to trace and compute slices. The full traces in each of these runs have over 2.1 million nodes; the simple pattern slices are almost as large, while the enriched pattern slices are only 95 nodes. For this example, slicing is over an order of magnitude faster using enriched patterns. The lazy tracing approach required less total time both for tracing and slicing (particularly for simple patterns). Thus, Haskell's built-in lazy evaluation strategy offers advantages by avoiding explicitly constructing the full trace in memory when it is not needed; however, there is still room for improvement. Again, however, for an actual workflow involving images or large data files, the evaluation time would be much larger, dwarfing the time for tracing or slicing.

Our implementation is a proof-of-concept that evaluates queries in-memory via interpretation, rather than compilation; further work would be needed to adapt our approach to support fine-grained provenance for conventional database systems. Nevertheless, our experimental results do suggest that the lazy tracing strategy and use of enriched patterns can effectively decrease the overhead of tracing, making it feasible for in-memory execution of workflows represented in NRC.

8. Related and future work

Program slicing has been studied extensively [18, 31, 32], as has the use of execution traces, for example in dynamic slicing. Our work contrasts with much of this work in that we regard the trace and underlying data as being of interest, not just the program. Some of our previous work [12] identified analogies between program slicing and provenance, but to our knowledge, there is no other prior work on slicing in databases.

Lineage and why-provenance were motivated semantically in terms of identifying *witnesses*, or parts of the input needed to ensure that a given part of the output is produced by a query. Early work on lineage in relational algebra [16] associates each output record with a witness. Buneman et al. studied a more general notion called why-provenance that maps an output part to a collection of witnesses [7, 8]. This idea was generalized further to the *how-provenance* or *semiring* model [19, 21], based on using algebraic expressions as annotations; this approach has been extended to handle some forms of negation and aggregation [4, 20]. Semiring homomorphisms commute with query evaluation; thus, homomorphic changes to the input can be performed directly on the output without re-running the query. However, this approach only applies to changes describable as semiring homomorphisms, such as deletion.

Where-provenance was also introduced by Buneman et al. [7, 8]. Although the idea of tracking where input data was copied from is natural, it is nontrivial to characterize semantically, because where-provenance does not always respect semantic equivalence. In later work, Buneman et al. [6] studied where-provenance for the pure NRC and characterized its expressiveness for queries and updates. It would be interesting to see whether their notion of *expressive completeness* for where-provenance could be extended to richer provenance models, such as traces, possibly leading to an implementation strategy via translation to plain NRC.

Provenance has been studied extensively for scientific workflow systems [5, 30], but there has been little formal work on the semantics of workflow provenance. The closest work to ours is that of Hidders et al. [22], who model workflows by extending the NRC with nondeterministic, external function calls. They sketch an operational semantics that records *runs* that contain essentially all of the information in a derivation tree, represented as a set of triples. They also suggest ways of extracting *subruns* from runs, but their treatment is partial and lacks strong formal guarantees analogous to our results.

There have been some attempts to reconcile the database and workflow views of provenance; Hidders et al. [22] argued for the use of Nested Relational Calculus (NRC) as a unifying formalism for both workflow and database operations, and subsequently Kwasnikowska and Van den Bussche [23] showed how to map this model to the Open Provenance Model. Acar et al. [2] later formalized a graph model of provenance for NRC. The most advanced work in this direction appears to be that of Amsterdamer et al. [3], who combined workflow and database styles of provenance in the context of the PigLatin system (a MapReduce variant based on nested relational queries). Lipstick allows analyzing the impact of restricted hypothetical changes (such as deletion) on parts of the output, but to our knowledge no previous work provides a formal guarantee about the impact of changes other than deletion.

In our previous work [12], we introduced *dependency provenance*, which conservatively over-approximates the changes that can take place in the output if the input is changed. We developed definitions and techniques for dependency provenance in full NRC including nonmonotone operations (`empty`, `sum`) and primitive functions. Dependency provenance cannot predict exactly how the output will be affected by a general modification to the source, but it can guarantee that some parts of the output will not change if certain parts of the input are fixed. Our notion of equivalence modulo a pattern is a generalization of the *equal-except-at* relation used in that work. Motivated by dependency provenance, an earlier technical report [10] presented a model of traced evaluation for NRC and proved elementary properties such as fidelity. However, it did not investigate slicing techniques, and used nondeterministic label generation instead of our deterministic scheme; our deterministic approach greatly simplifies several aspects of the system, particularly for slicing.

There are several intriguing directions for future work, including developing more efficient techniques for traced evaluation and slicing that build upon existing database query optimization capabilities. It appears possible to translate multiset queries so as to make the labels explicit, since a fixed given query increases the label depth by at most a constant. Thus, it may be possible to evaluate queries with label information but without tracing first, then gradually build the trace by slicing backwards through the query, re-evaluating subexpressions as necessary. Other interesting directions include the use of slicing techniques for security, to hide confidential input information while disclosing enough about the trace to permit recomputation, and the possibility of extracting other forms of provenance from traces, as explored in the context of functional programs in prior work [1].

9. Conclusion

The importance of provenance for transparency and reproducibility is widely recognized, yet there has been little explicit discussion of correctness properties formalizing intuitions about how provenance is to provide reproducibility. In self-explaining computation, traces are considered to be explanations of a computation in the sense that the trace can be used to recompute (parts of) the output under hypothetical changes to the input. This paper develops the foundations of self-explaining computation for database queries, by defining a tracing semantics for NRC, proposing a formal definition of correctness for tracing (fidelity) and slicing, and defining a correct (though potentially overapproximate) algorithm for trace

slicing. Trace slicing can be used to obtain smaller "golden trail" traces that explain only a part of the input or output, and explore the impact of changes in hypothetical scenarios similar to the original run. At a technical level, the main contributions are the careful use of prefix codes to label multiset elements, and the development of enriched patterns that allow more precise slices. Our design is validated by a proof-of-concept implementation that shows that laziness and enriched patterns can significantly improve performance for small (in-memory) examples.

In the near term, we plan to combine our work on self-explaining functional programs [28] and database queries (this paper) to obtain slicing and provenance models for programming languages with query primitives, such as F# [15] or Links [24]. Ultimately, our aim is to extend self-explaining computation to programs that combine several execution models, including workflows, databases, conventional programming languages, Web interaction, or cloud computing.

Acknowledgments We are grateful to Peter Buneman, Jan Van den Bussche, and Roly Perera for comments on this work and to the anonymous reviewers for detailed suggestions. Effort sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-13-1-3006. The U.S. Government and University of Edinburgh are authorized to reproduce and distribute reprints for their purposes notwithstanding any copyright notation thereon. Cheney is supported by a Royal Society University Research Fellowship, by the EU FP7 DIACHRON project, and EPSRC grant EP/K020218/1. Acar is partially supported by an EU ERC grant (2012-StG 308246—DeepSea) and an NSF grant (CCF-1320563).

References

[1] U. A. Acar, A. Ahmed, J. Cheney, and R. Perera. A core calculus for provenance. *Journal of Computer Security*, 21:919–969, 2013. Full version of a POST 2012 paper.

[2] U. A. Acar, P. Buneman, J. Cheney, N. Kwasnikowska, J. Van den Bussche, and S. Vansummeren. A graph model of data and workflow provenance. In *TAPP*, 2010.

[3] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *PVLDB*, 5(4):346–357, 2011.

[4] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS*, pages 153–164. ACM, 2011.

[5] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

[6] P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems*, 33(4):28, November 2008.

[7] P. Buneman, S. Khanna, and W. Tan. Why and where: A characterization of data provenance. In *ICDT*, number 1973 in LNCS, pages 316–330. Springer, 2001.

[8] P. Buneman, S. Khanna, and W. Tan. On propagation of deletions and annotations through views. In *PODS*, pages 150–158, 2002.

[9] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comp. Sci.*, 149(1):3–48, 1995.

[10] J. Cheney, U. A. Acar, and A. Ahmed. Provenance traces. *CoRR*, arXiv.org/abs/0812.0564, 2008.

[11] J. Cheney, U. A. Acar, and R. Perera. Toward a theory of self-explaining computation. In *In search of elegance in the theory and practice of computation: a Festschrift in honour of Peter Buneman*, number 8000 in LNCS, pages 193–216. Springer, 2013.

[12] J. Cheney, A. Ahmed, and U. A. Acar. Provenance as dependency analysis. *Mathematical Structures in Computer Science*, 21(6):1301–1337, 2011.

[13] J. Cheney, A. Ahmed, and U. A. Acar. Database queries that explain their work (extended version). Technical report, arXiv.org, 2014. `http://arxiv.org/abs/1408.1675`.

[14] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[15] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, pages 403–416, New York, NY, USA, 2013. ACM.

[16] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[17] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[18] J. Field and F. Tip. Dynamic dependence in term rewriting systems and its application to program slicing. *Information and Software Technology*, 40(11–12):609–636, 1998.

[19] J. N. Foster, T. J. Green, and V. Tannen. Annotated XML: queries and provenance. In *PODS*, pages 271–280, 2008.

[20] F. Geerts and A. Poggi. On database query languages for K-relations. *J. Applied Logic*, 8(2):173–185, 2010.

[21] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[22] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. Van den Bussche. A formal model of dataflow repositories. In *DILS*, 2007.

[23] N. Kwasnikowska and J. Van den Bussche. Mapping the NRC dataflow model to the open provenance model. In *IPAW*, pages 3–16, 2008.

[24] S. Lindley and J. Cheney. Row-based effect types for database integration. In *TLDI*, pages 91–102. ACM Press, 2012.

[25] P. Missier, B. Ludäscher, S. Dey, M. Wang, T. McPhillips, S. Bowers, M. Agun, and I. Altintas. Golden trail: Retrieving the data history that matters from a comprehensive provenance repository. *International Journal of Digital Curation*, 7(1):139–150, 2011.

[26] L. Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2–3), 2010.

[27] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110. ACM, 2008.

[28] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *ICFP*, pages 365–376. ACM, 2012.

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

[30] Y. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.

[31] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[32] M. Weiser. Program slicing. In *ICSE*, pages 439–449. IEEE Press, 1981.