On the Expressiveness of Implicit Provenance in Query and Update Languages

Peter Buneman[1]*, James Cheney[1]*, and Stijn Vansummeren[2]**

[1] University of Edinburgh, Scotland
[2] Hasselt University and Transnational University of Limburg, Belgium

Abstract. Information concerning the origin of data (that is, its *provenance*) is important in many areas, especially scientific recordkeeping. Currently, provenance information must be maintained explicitly, by added effort of the database maintainer. Since such maintenance is tedious and error-prone, it is desirable to provide support for provenance in the database system itself. In order to provide such support, however, it is important to provide a clear explanation of the behavior and meaning of existing database operations, both queries and updates, with respect to provenance. In this paper we take the view that a query or update *implicitly* defines a provenance mapping linking components of the output to the originating components in the input. Our key result is that the proposed semantics are expressively complete relative to natural classes of queries that *explicitly* manipulate provenance.

1 Introduction

The *provenance* of data – its origins and how it came to be included in a database – has recently sparked some research interest [4, 12, 14]. The topic is particularly important in those scientific databases, sometimes referred to as *curated* databases, that are constructed by a labor-intensive process of copying, correcting and annotating data from other sources. The value of curated databases lies in their organization and in the trustworthiness of their data. Provenance is particularly important in assessing the latter. In practice, provenance – if it is recorded at all – is recorded manually, which is both time-consuming and error-prone. Automated provenance recording support is desirable, and for this it is essential to have a proper semantic foundation to guide us on what should be recorded and to understand what effect database operations have on provenance.

We focus on a specific kind of provenance associated with the copying and modification of data by query and update languages. We use a formalization based on the "tagging" or "propagation" approach of Wang and Madnick [15] and Bhagwat et al. [3]. In this approach, it is assumed that each input data item has an identifying color. Existing database operations are then given a new semantics as functions mapping such colored databases to colored databases in

* Supported by the UK EPSRC (Digital Curation) and the Royal Society.
** Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

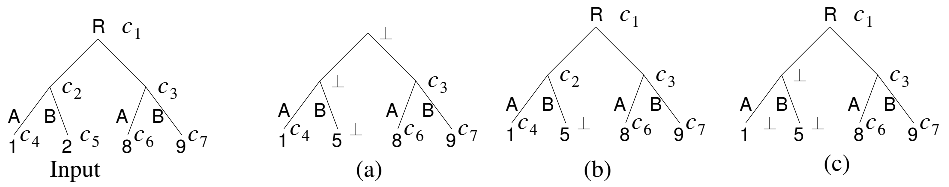

Fig. 1. Color propagation for query (a) and updates (b) and (c).

which colors are propagated along with their data item during computation of the output. The provenance of a data item in the output is then simply that input data item with the same color. To illustrate this approach, consider a table $R(A, B)$ with tuples $\{(1,2),(8,9)\}$, and consider the following SQL query.

$$\begin{array}{c}(\texttt{select * from } R \texttt{ where } A \texttt{ <> } 1) \\ \texttt{union } (\texttt{select } A\texttt{, } 5 \texttt{ as } B \texttt{ from } R \texttt{ where } A = 1)\end{array} \qquad (a)$$

The input tree at the left of Fig. 1 is a representation of R in which the atomic data values, the tuples and the table R itself are all annotated with colors $c_1, c_2, \ldots$. We could then define the colored semantics of query (a) to map the input to the colored table represented by the tree (a) in Fig. 1. This defines the provenance of the atom 1 in the output to be the corresponding atom in R, the provenance of the tuple $(8, 9)$ to be the second tuple in R, and so on. The color $\perp$ indicates that a data item is introduced by the query itself. Hence, this particular colored semantics takes the view that queries construct new tables and that the second `select` subquery constructs a new tuple rather than copying an existing one.

Color-propagating functions from colored databases to colored databases can hence be used to formally define the provenance behavior of existing database operations. By "color-propagating" we mean that the function should only use colors to indicate the origin of output items: if the function is applied to a recolored version of the input, then it should produce an output with the same recoloring applied. In particular, the input colors cannot influence the uncolored part of the output and the function's behavior is insensitive to the actual choice of colors used in the input. We shall refer to such propagating functions as *provenance-aware operations.*

The particular provenance ascribed to query (a) in Fig. 1 has the property that if an output item j has the same color as an input item i, then i and j are identical. We shall call provenance-aware operations with this property *copying.* A copying operation has the property that if some item is colored $\perp$ ("blank"), all items that contain it will also be colored $\perp$.

The provenance of query (a) described in Fig. 1 is exactly the "intuitive" or "default" provenance of SQL queries proposed by Bhagwat et al. [3], although they only consider provenance of atomic values. In particular, the default provenance is always copying, as it views constant and tuple constructors in queries as creating new items. Of course, this default semantics may not be the provenance semantics that a curator wants to give to a particular query. For this reason, [3] proposes an extension to query languages that allows provenance to be defined explicitly. Our first result is to propose a default provenance semantics for query languages, similar to that given in [15] and [3], and show that it is *complete* in the sense that it expresses exactly the explicitly definable provenance-aware operations that are copying. This shows that the default provenance semantics is a reasonable semantics for queries.

Turning to updates, we note that simple update languages, such as the expressions of SQL that modify data, do not express more database transformations than do SQL queries, and they have been largely ignored in database theory. However, the following examples show that the story is very different when we take account of provenance.

```
update R set B = 5 where A = 1                                 (b)
delete from R where A = 1; insert into R values (1,5)          (c)
```

Since updates do not construct new databases, but modify existing ones in-place, it is reasonable to define their provenance semantics in a way that agrees with how tuple identifiers are preserved in practical database management systems. For example, the provenance of updates (b) and (c) would behave on R as illustrated in Fig. 1(b) and Fig. 1(c), respectively. Note that this provenance semantics is no longer copying. For example, the provenance of the tuple $(1, 5)$ in Fig. 1(b) is the tuple $(1, 2)$ from the input, although they are clearly not identical. For this reason we introduce a weaker semantic restriction on provenance-aware operations, and consider the *kind-preserving* ones. By "kind-preserving" we mean that if output item j has the same color as input item i, then they are of the same *kind*: they are both sets, both tuples, or identical atoms. Kind-preserving operations allow the output type of an item to differ from its input type, and this is practically important in considering operations such as SQL's `add column` update, which extends a tuple but does not change its provenance.

We propose a default provenance semantics for updates as kind-preserving operations, and show this semantics to be complete in the sense that every explicitly definable kind-preserving provenance-aware operation can be expressed by the default provenance semantics of an update.

Most previous work on provenance focuses on the relational model. We shall work in the more general "nested relational" or complex object data model [1, 7] for two reasons. First, as our examples indicate, we are interested in provenance at all levels: atoms, tuples, and tables (sets of tuples); a complex object model allows us to provide a uniform treatment of these levels. Second, complex object models are widely used in scientific data, where provenance is of paramount importance. Liefke and Davidson [11] proposed a simple and elegant language that extends SQL-style updates to complex objects. To be more precise about our completeness result for updates: it is the default provenance of this language that we show complete with regard to the explicitly definable, kind-preserving provenance-aware operations. It is therefore a natural choice for updating complex-object databases when one wants to record provenance.

Related work. There is a substantial body of research on provenance (sometimes termed *lineage* or *pedigree*) in both database and scientific computing settings, which is nicely surveyed in [4, 12, 14]. In early approaches to provenance [15, 8] the provenance of an output tuple consists of sets of input tuples that directly or indirectly influenced the output. These techniques only track the provenance of tuples in relational data. In [6] a distinction is made between "why" and "where" provenance for queries in a tree-structured model. More recently, [5] investigated tracking where-provenance for manual updates to curated databases. The Trio project [2] has investigated the combination of tuple-level lineage with uncertainty and accuracy information.

There has also been significant work on the properties of "tagging" or "annotation" in databases. Tan [13] studied theoretical issues of query containment and equivalence in the presence of annotations. The DBNotes system [3] uses variations on why- and where-provenance to propagate annotations on source data through queries. Geerts et al. have developed Mondrian [10], a database system that supports *block annotations*, in which a color can be associated with a subset of the fields in a table, not just a single value.

Finally, provenance has also been studied in the geospatial and Grid computing communities [4, 9, 12]. Here, the motivation is to record the workflow that constructs large data sets in order to avoid repeated computation.

2 Preliminaries

Let us first sketch the languages used throughout this paper. As query languages, we employ the nested relational algebra $\mathcal{NRA}$ and the nested relational calculus $\mathcal{NRC}$ [7]. We also use the nested update language $\mathcal{NUL}$, based on the complex object update language $\mathcal{CUCA}$ [11], which generalizes familiar SQL updates to complex objects. All of these languages deal with complex objects in the form of nested relations, whose types are given by the following grammar:

$$s, t ::= b \mid s \times t \mid \{s\}.$$

Here, b ranges over some unspecified finite collection of base types like the booleans, the integers, and so on. We assume this collection to include at least the special base type *unit*. Types denote sets of *objects*. The type *unit* consists only of the empty tuple $()$; objects of $s \times t$ are pairs (v, w) with v and w objects of type s and t, respectively; and objects of $\{s\}$ are finite sets of objects, each of type s. We write $v\colon s$ to indicate that v is an object of type s. Furthermore, we feel free to omit parentheses and write $s_1 \times \cdots \times s_n$ for $(\ldots((s_1 \times s_2) \times s_3) \cdots \times s_n)$. Our results hold if we use labeled records instead of pairs; but the syntax of pairs is more manageable.

The expressions of $\mathcal{NRA}$, $\mathcal{NRC}$, and $\mathcal{NUL}$ are explicitly typed and are formed using the typing rules of Fig. 2. Here, we range over $\mathcal{NRA}$ expressions by f, g, and h; over $\mathcal{NRC}$ expressions by e; and over $\mathcal{NUL}$ expressions by u. We will often omit the explicit type annotations in superscript when they are clear from the context.

EXPRESSIONS OF $\mathcal{NRA}$.

$$\frac{}{Ka\colon\ unit \to b} \qquad \frac{}{id^s\colon s \to s} \qquad \frac{h\colon r \to s \quad g\colon s \to t}{g \circ h\colon r \to t}$$

$$\frac{}{!^s\colon s \to unit} \qquad \frac{}{\pi_1^{s,t}\colon s \times t \to s} \qquad \frac{}{\pi_2^{s,t}\colon s \times t \to t} \qquad \frac{h\colon r \to s \quad g\colon r \to t}{\langle g, h\rangle\colon r \to s \times t}$$

$$\frac{}{\eta^s\colon s \to \{s\}} \qquad \frac{}{\mu^s\colon \{\{s\}\} \to \{s\}} \qquad \frac{}{K\{\}^s\colon\ unit \to \{s\}} \qquad \frac{}{\cup^s\colon \{s\} \times \{s\} \to \{s\}}$$

$$\frac{}{\rho_2^{s,t}\colon s \times \{t\} \to \{s \times t\}} \qquad \frac{f\colon s \to t}{map(f)\colon \{s\} \to \{t\}} \qquad \frac{}{cond^t\colon s \times s \times t \times t \to t}$$

EXPRESSIONS OF $\mathcal{NRC}$.

$$\frac{}{a\colon b} \qquad \frac{}{x^s\colon s} \qquad \frac{e\colon t}{\lambda x^s.e\colon s \to t} \qquad \frac{e_1\colon s \to t \quad e_2\colon s}{e_1\, e_2\colon t}$$

$$\frac{}{()\colon\ unit} \qquad \frac{e\colon s \times t}{\pi_1\, e\colon s \quad \pi_2\, e\colon t} \qquad \frac{e_1\colon s \quad e_2\colon t}{(e_1, e_2)\colon s \times t} \qquad \frac{}{\{\}^s\colon \{s\}} \qquad \frac{e\colon s}{\{e\}\colon \{s\}}$$

$$\frac{e_1\colon \{s\} \quad e_2\colon \{s\}}{e_1 \cup e_2\colon \{s\}} \qquad \frac{e_1\colon \{s\} \quad e_2\colon \{t\}}{\bigcup\{e_2 \mid x^s \in e_1\}\colon \{t\}} \qquad \frac{e_1\colon s \quad e_2\colon s \quad e_3\colon t \quad e_4\colon t}{\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4\colon t}$$

EXPRESSIONS OF $\mathcal{NUL}$.

$$\frac{}{\mathsf{skip}^s\colon s \to s} \qquad \frac{u_1\colon r \to s \quad u_2\colon s \to t}{u_1; u_2\colon r \to t} \qquad \frac{e\colon t}{\mathsf{repl}^s\ e\colon s \to t} \qquad \frac{u\colon s \to t}{[x^s]\, u\colon s \to t}$$

$$\frac{e\colon \{s\}}{\mathsf{insert}\, e\colon \{s\} \to \{s\}} \qquad \frac{e\colon \{s\}}{\mathsf{remove}\, e\colon \{s\} \to \{s\}} \qquad \frac{u\colon s \to t}{\mathsf{iter}\ u\colon \{s\} \to \{t\}}$$

$$\frac{u\colon r \to t}{\mathsf{updl}^s\ u\colon r \times s \to t \times s} \qquad \frac{u\colon s \to t}{\mathsf{updr}^r\ u\colon r \times s \to r \times t}$$

Fig. 2. Expressions of $\mathcal{NRL}$.

Semantics of $\mathcal{NRA}$. The $\mathcal{NRA}$ is an algebra of functions over complex objects. Every $\mathcal{NRA}$ expression $f\colon s \to t$ defines a function from s to t. The expression Ka is the constant function that always produces the atom a; id is the identity function; and $g \circ h$ is function composition, i.e., $(g \circ h)(v) = g(h(v))$. For pairs: ! produces () on all inputs; π_1 and π_2 are respectively the left and right projections; and $\langle g, h\rangle$ is pair formation: $\langle g, h\rangle(v) = (g\, v, h\, v)$. For sets: η forms singletons: $\eta(v) = \{v\}$; $K\{\}$ is the constant function that produces the empty set; $\cup$ is set union; μ flattens sets of sets: $\mu(\{V, \ldots, V'\}) = V \cup \cdots \cup V'$; ρ_2 is the right tensor product: $\rho_2(v, \{w, \ldots, w'\}) = \{(v, w), \ldots, (v, w')\}$; and $map(f)$ applies f to every object in its input set: $map(f)(\{v, \ldots, v'\}) = \{f(v), \ldots, f(v')\}$. Finally, $cond$ is the conditional that, when applied to a tuple (v, v', w, w') returns w if $v = v'$, and returns w' otherwise.

Example 1. Here are some simple examples of the functions that are definable in $\mathcal{NRA}$. The relational projections $\Pi_1 \colon \{s \times t\} \to \{s\}$ and $\Pi_2 \colon \{s \times t\} \to \{t\}$ on sets of pairs are given by $\Pi_1 := map(\pi_1)$ and $\Pi_2 := map(\pi_2)$, respectively. The tensor product ρ_1 similar to ρ_2 but pairing to the left is defined as $\rho_1 := map(\langle \pi_2, \pi_1 \rangle) \circ \rho_2 \circ \langle \pi_2, \pi_1 \rangle$. Cartesian product of two sets is then readily defined as $cartprod := \mu \circ map(\rho_1) \circ \rho_2$.

Semantics of $\mathcal{NRC}$. The semantics of $\mathcal{NRC}$ is that of the first-order, simply typed lambda calculus with products and sets. As such, expression a denotes the constant a; x^s is the explicitly typed variable that can be bound to objects of type s; $\lambda x.e$ is standard lambda abstraction; and $e_1\, e_2$ is function application. Furthermore, expression $()$ denotes the empty tuple; (e_1, e_2) is pair construction; and $\pi_1\, e$ and $\pi_2\, e$ are respectively the left and right projection on pairs. For sets: $\{\}$ denotes the empty set; $\{e\}$ is singleton construction; $e_1 \cup e_2$ is set union; and $\bigcup\{e_2 \mid x \in e_1\}$ is set comprehension. That is, $\bigcup\{e_2 \mid x \in e_1\} = f(v) \cup \cdots \cup f(v')$ where $f = \lambda x.e_2$ and e_1 denotes $\{v, \ldots, v'\}$. Finally, $\mathsf{if}\ e_1 = e_2 \mathsf{then}\ e_3\ \mathsf{else}\ e_4$ is the conditional expression that returns e_3 if the denotations of e_1 and e_2 are equal and returns e_4 otherwise.

Example 2. The left relational projection $\Pi_1 \colon \{s \times t\} \to \{s\}$ on a sets of pairs is defined in $\mathcal{NRC}$ as $\lambda U. \bigcup\{\{\pi_1\, x\} \mid x \in U\}$. Right relational projection is defined similarly. SQL query (a) from the Introduction is defined in $\mathcal{NRC}$ as $e_{(a)} := \bigcup\{\mathsf{if}\ \pi_1\, x = 1\ \mathsf{then}\ \{(\pi_1\, x, 5)\}\ \mathsf{else}\ \{y\} \mid y \in R\}$. Here, the table $R(A, B)$ is represented as a set of pairs $R \colon \{b \times b\}$. Finally the expression,

$$\bigcup\{\bigcup\{\mathsf{if}\ \pi_1\, x = \pi_1\, y\ \mathsf{then}\ \{((\pi_1\, x, \pi_2\, x), \pi_2\, y)\}\ \mathsf{else}\ \{\} \mid y \in S\} \mid x \in R\}$$

defines the relational join of two sets of pairs $R \colon \{r \times s\}$ and $S \colon \{r \times t\}$.

We note that the power of $\mathcal{NRC}$ is not restricted to simple select-project-join queries. It is well-known that the conditional expression allows definition of all other non-monotone operations such as difference, intersection, set membership testing, subset testing, and nesting [7]. Furthermore,

Proposition 1 ([7]). *$\mathcal{NRA} \equiv \mathcal{NRC}$ in the sense that every function definable by an expression $f \colon s \to t$ in $\mathcal{NRA}$ is definable by a closed expression $e \colon s \to t$ in $\mathcal{NRC}$, and vice versa.*

Semantics of $\mathcal{NUL}$. Note that most $\mathcal{NUL}$ updates syntactically contain $\mathcal{NRC}$ expressions. Each $\mathcal{NUL}$ update $u \colon s \to t$ defines a function that intuitively modifies objects of type s "in-place" to objects of type t. First, we have some "control" updates: skip is the trivial update with $\mathsf{skip}(v) = v$; while $u_1; u_2$ is update composition: $(u_1; u_2)(v) = u_2(u_1(v))$. The expression $\mathsf{repl}\, e$ replaces the input object by the object denoted by e. Next, $[x]\, u$ binds all free occurrences of x in $\mathcal{NRC}$ expressions occurring in u to the input object and then performs u. For example, $([x]\, \mathsf{repl}\, (x, x))(v) = (v, v)$. Note that the value of x is immutable; it is not affected by the changes u makes to the input object. In particular,

$[x]\,\big(\mathsf{repl}();\mathsf{repl}(x,x)\big)$ is equivalent to $[x]\,\mathsf{repl}(x,x)$. Next come the set updates: $(\mathsf{insert}\ e)(V) = V \cup W$ where W is the denotation of e; $(\mathsf{remove}\,e)(V) = V - W$ where W is the denotation of e; and $\mathsf{iter}\ u$ applies u to every object in its input: $(\mathsf{iter}\ u)(\{v, \dots, v'\}) = \{u(v), \dots, u(v')\}$. Finally we have the updates on pairs: $(\mathsf{updl}\,u)(v,w) = (u(v), w)$ and $(\mathsf{updr}\,u)(v,w) = (v, u(w))$.

Example 3. We express the SQL updates (b) and (c) from the Introduction in $\mathcal{NUL}$. Here, the table $R(A,B)$ is represented as an object $R\colon \{b \times b\}$, which serves as the context object for the $\mathcal{NUL}$ updates. Example (b) is expressed as $u_{(b)} := \mathsf{iter}\,\big([x]\,\mathsf{updr}\ \mathsf{repl}(\mathsf{if}\ \pi_1\,x = 1\ \mathsf{then}\ 5\ \mathsf{else}\ \pi_2\,x)\big)$. Example ($c$) is expressed as $u_{(c)} := [x]\,\mathsf{remove} \bigcup\{\mathsf{if}\ \pi_1\,y = 1\ \mathsf{then}\ \{y\}\ \mathsf{else}\ \{\} \mid y \in x\};\ \mathsf{insert}\,\{(1,5)\}$. We can also express schema modifying updates such as `alter table` R `drop column` B that transforms $R\colon \{b \times b\}$ into $R\colon \{b\}$ in $\mathcal{NUL}$ by $\mathsf{iter}\,([x]\,\mathsf{repl}\,\pi_1\,x)$.

Theorem 1. *$\mathcal{NRA}$, $\mathcal{NRC}$, and $\mathcal{NUL}$ are all equally expressive.*

Hence, we may view expressions in each of the three languages as "syntactic sugar" for expressions in the other languages. This allows us to freely combine $\mathcal{NRA}$, $\mathcal{NRC}$, and $\mathcal{NUL}$ into the single nested relational language $\mathcal{NRL}$.

3 A Model of Provenance

In this section we begin our study of provenance. Let *color* be an additional base type (not included in the unspecified collection of base types of $\mathcal{NRL}$) whose infinite set of elements we will refer to as *colors*. Let the *color-extended types* be the types in which *color* may also occur:

$$\mathbf{s}, \mathbf{t} := \mathit{color} \mid b \mid \mathbf{s} \times \mathbf{t} \mid \{\mathbf{s}\}.$$

To avoid possible confusion, $\mathbf{r}, \mathbf{s}$ and $\mathbf{t}$ will range over color-extended types and r, s and t over ordinary $\mathcal{NRL}$ types. Let $\mathbf{s} * \mathbf{t}$ be the type of objects of type $\mathbf{s}$ that are recursively paired with objects of type $\mathbf{t}$:

$$\mathit{color}{*}\mathbf{t} := \mathit{color} \times \mathbf{t} \quad b{*}\mathbf{t} := b{\times}\mathbf{t} \quad (\mathbf{r} \times \mathbf{s}){*}\mathbf{t} := (\mathbf{r}{*}\mathbf{t}{\times}\mathbf{s}{*}\mathbf{t}){\times}\mathbf{t} \quad \{\mathbf{s}\}{*}\mathbf{t} := \{\mathbf{s}{*}\mathbf{t}\}{\times}\mathbf{t}$$

We then define the type $\underline{s}$ of *colored objects of type s* as $s * \mathit{color}$. A colored object is hence an object in which each subobject is paired with a color. Let $\perp$ be a special color that describes the provenance of newly created objects. A *distinctly colored* object is a colored object in which $\perp$ does not occur and in which each other color occurs at most once.

As we have already illustrated in the Introduction, we can describe the provenance behavior of database operations by color-propagating functions from distinctly colored objects to colored objects. For our further formalisation it is more convenient, however, to consider color-propagating functions $f\colon \underline{s} \to \underline{t}$ that operate on *all* colored objects. Here, *color-propagating* means that f cannot let input colors influence the uncolored part of the output and that f's behavior is insensitive to the actual colors used in the input. In particular, a function

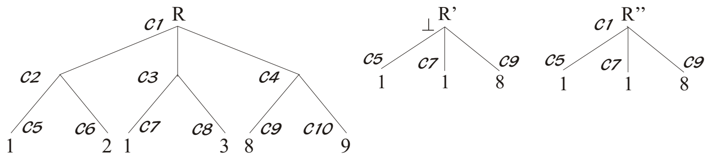

Fig. 3. The provenance semantics of left relational projection.

$g\colon \underline{b \times b} \to \underline{b}$ that outputs $(1, \text{red})$ when its two input atoms are colored equally, but outputs $(2, \text{blue})$ otherwise is not color-propagating. Formally, we require that $f \circ \alpha^*_{\underline{s}} = \alpha^*_{\underline{t}} \circ f$ for any "recoloring" $\alpha\colon \mathit{color} \to \mathit{color}$ that maps $\perp$ to $\perp$. Here, $\alpha^*_{\underline{r}}\colon \underline{r} \to \underline{r}$ is the canonical extension of α to type $\underline{r}$:

$$\alpha^*_{\underline{b}} := \mathit{id} \times \alpha \qquad \alpha^*_{\underline{r \times r'}} := (\alpha^*_{\underline{r}} \times \alpha^*_{\underline{r'}}) \times \alpha \qquad \alpha^*_{\underline{\{r\}}} := \mathit{map}(\alpha^*_{\underline{r}}) \times \alpha,$$

where $h \times h'$ is an abbreviation of $\langle h \circ \pi_1, h' \circ \pi_2 \rangle$. Note that "color-propagating" is a different concept than "generic w.r.t. colors" since α above is not required to be bijective. Also note that this definition ensures that all colors in $f(v)$, except $\perp$, also occur in v. Finally, note that the behavior of f is fully determined by its behavior on distinctly colored objects, as the following lemma shows.

Lemma 1. *If $f\colon \underline{s} \to \underline{t}$ and $g\colon \underline{s} \to \underline{t}$ are two color-propagating functions such that $f(v) = g(v)$ for each distinctly colored $v\colon \underline{s}$, then $f \equiv g$.*

Proof. Let $w\colon \underline{s}$ be arbitrary and fix some distinctly colored $v\colon \underline{s}$ that equals w modulo colors. Then there obviously exists some recoloring α such that $\alpha^*_{\underline{s}}(v) = w$. Hence, $f(w) = f(\alpha^*_{\underline{s}}(v)) = \alpha^*_{\underline{t}}(f(v)) = \alpha^*_{\underline{t}}(g(v)) = g(\alpha^*_{\underline{s}}(v)) = g(w)$. ☐

Database operations are typically "domain-preserving" and are hence limited in their ability to create new atomic data values. In particular, if $o\colon s \to t$ is a query or update that creates atom a (in the sense that a appears in $o(v)$ but not in v for some v), then a appears as a constant in o. We want our concept of "provenance-aware operation" to reflect this behavior. We therefore define $f\colon \underline{s} \to \underline{t}$ to be *bounded-inventing* if there exists a finite set A of atoms such that for every distinctly colored $v\colon \underline{s}$ and every $(a, c)\colon \underline{b}$ occurring in $f(v)$, if f says that it created a (i.e., if $c = \perp$), then $a \in A$.

Definition 1. *A* provenance-aware database operation *(pado for short) is a color-propagating, bounded-inventing function $f\colon \underline{s} \to \underline{t}$.*

It is important to note that a pado may define an object in the output to come from multiple parts in the input. For example, we will define the provenance semantics of the left relational projection Π_1 such that it maps the colored object $R\colon \underline{\{b \times b\}}$ from Fig. 3 to R' in that figure. Note that atom 1 originated from both the first and the second pair in the input, as it appears both with colors c_5 and c_7 in R'.

In what follows, we will consider two natural classes of pados: copying and kind-preserving. Intuitively, a pado is *copying* if every object in the output that was not created by f was copied verbatim from the input.

Definition 2 (Copying). *A pado* $f\colon \underline{s} \to \underline{t}$ *is copying if for every* $v\colon \underline{s}$ *and every colored subobject* $(w, c)\colon \underline{r}$ *of* $f(v)$*, if* $c \neq \perp$ *then* (w, c) *occurs in* v*.*

Similarly, a pado is kind-preserving if every subobject in the output that was not created by f originates from an object in the input of the same kind. In particular, every copying pado is also kind-preserving.

Definition 3 (Kind-preserving). *A pado* $f\colon \underline{s} \to \underline{t}$ *is* kind-preserving *if for every* $v\colon \underline{s}$ *and every colored subobject* $(w, c)\colon \underline{r}$ *of* $f(v)$*, if* $c \neq \perp$ *then there exists* (u, c) *in* v *such that* u *and* v *are of the same kind: they are both sets, both pairs, or the same atom.*

Define $\mathcal{NRL}(\mathit{color})$ to be the extension of $\mathcal{NRL}$ with the base type *color* in which $\perp$ is the only color that may appear as a constant. Since $\mathcal{NRL}(\mathit{color})$ can explicitly manipulate colors, it is a natural language for the "explicit" definition of pados and a suitable benchmark to compare proposals for "standard" provenance semantics of query and update languages against. Define $\mathcal{CP}$ and $\mathcal{KP}$ as the sets of closed expressions in $\mathcal{NRL}(\mathit{color})$ defining respectively copying and kind-preserving pados:

$$\mathcal{CP} := \{\ f \mid f\colon \underline{s} \to \underline{t} \text{ in } \mathcal{NRL}(\mathit{color}) \text{ defines a copying pado }\},$$
$$\mathcal{KP} := \{\ f \mid f\colon \underline{s} \to \underline{t} \text{ in } \mathcal{NRL}(\mathit{color}) \text{ defines a kind-preserving pado }\}.$$

Note that $\mathcal{CP}$ and $\mathcal{KP}$ are semantically defined. In fact both $\mathcal{CP}$ and $\mathcal{KP}$ are undecidable: a standard reduction from the satisfiability problem of the relational algebra shows that checking if an $\mathcal{NRL}(\mathit{color})$ expression is color-propagating, bounded-inventing, copying, or kind-preserving are all undecidable.

4 Provenance for Query Languages

In this section we give an intuitive provenance-aware semantics for $\mathcal{NRA}$ and $\mathcal{NRC}$ expressions. Concretely, we take the view that queries construct new objects. As such, all objects constructed by a constant, pair, or set constructor (including union and map/comprehension) during a query are colored $\perp$. Objects copied from the input retain their color. The provenance semantics $\mathcal{P}[f]\colon \underline{s} \to \underline{t}$ of an $\mathcal{NRA}$ expression $f\colon s \to t$ is formally defined in Fig. 4 by translation into $\mathcal{NRL}(\mathit{color})$. There, we write $(g \times h)$ as a shorthand for $\langle g \circ \pi_1, h \circ \pi_2\rangle$; $\perp$ as a shorthand for $K\perp \circ !$; $\mathit{\Pi}_1$ as a shorthand for the left relational projection $\mathit{map}(\pi_1)$; and $\mathit{val}_{\underline{s}}\colon \underline{s} \to s$ for the function that forgets colors:

$$\mathit{val}_{\underline{b}} := \pi_1 \qquad \mathit{val}_{\underline{s \times t}} := (\mathit{val}_{\underline{s}} \times \mathit{val}_{\underline{t}}) \circ \pi_1 \qquad \mathit{val}_{\underline{\{s\}}} := \mathit{map}(\mathit{val}_{\underline{s}}) \circ \pi_1.$$

Note that $\mathcal{P}[\mathit{cond}]$ ignores colors during comparison: applied to a colored tuple $((v, v', w, w'), c)$ it returns w if $\mathit{val}(v) = \mathit{val}(v')$, and w' otherwise.

The provenance semantics $\mathcal{P}[e]\colon \underline{s}$ and $\mathcal{P}[e']\colon \underline{s} \to \underline{t}$ of $\mathcal{NRC}$ expressions $e\colon s$ and $e'\colon s \to t$ is also defined in Fig. 4 by translation into $\mathcal{NRL}(\mathit{color})$.

PROVENANCE SEMANTICS OF $\mathcal{NRA}$.

$$\begin{array}{rlrlrl}
\mathcal{P}[Ka] & := Ka \times \bot & \mathcal{P}[id^s] & := id^{\underline{s}} & \mathcal{P}[g \circ h] & := \mathcal{P}[g] \circ \mathcal{P}[h] \\
\mathcal{P}[!] & := ! \times \bot & \mathcal{P}[\pi_1] & := \pi_1 \circ \pi_1 & \mathcal{P}[\pi_2] & := \pi_2 \circ \pi_1 \\
\mathcal{P}[\langle g, h \rangle] & := (\mathcal{P}[g] \times \mathcal{P}[h]) \times \bot & \mathcal{P}[\eta] & := \eta \times \bot & \mathcal{P}[\mu] & := \mu \circ \mathit{\Pi}_1 \times \bot \\
\mathcal{P}[K\{\}] & := K\{\} \times \bot & \mathcal{P}[\cup] & := \cup \times \bot & \mathcal{P}[\rho_2] & := \rho_2 \times \bot \\
\mathcal{P}[map(f)] & := map(\mathcal{P}[f]) \times \bot & \mathcal{P}[cond] & \multicolumn{3}{l}{:= cond \circ (val \times val \times id \times id) \circ \pi_1}
\end{array}$$

PROVENANCE SEMANTICS OF $\mathcal{NRC}$.

$$\begin{array}{rlrl}
\mathcal{P}[a] & := (a, \bot) & \mathcal{P}[\lambda x^s . e] & := \lambda x^{\underline{s}} . \mathcal{P}[e] \\
\mathcal{P}[x^s] & := x^{\underline{s}} & \mathcal{P}[e_1\, e_2] & := \mathcal{P}[e_1]\, \mathcal{P}[e_2] \\
\mathcal{P}[()] & := \big((), \bot\big) & \mathcal{P}[\pi_1\, e] & := \pi_1\, \pi_1\, \mathcal{P}[e] \\
\mathcal{P}[\pi_2\, e] & := \pi_2\, \pi_1\, \mathcal{P}[e] & \mathcal{P}[(e_1, e_2)] & := \big((\mathcal{P}[e_1], \mathcal{P}[e_2]), \bot\big) \\
\mathcal{P}[\{\}] & := (\{\}, \bot) & \mathcal{P}[e_1 \cup e_2] & := \big((\pi_1\, \mathcal{P}[e_1] \cup \pi_1\, \mathcal{P}[e_2]), \bot\big) \\
\mathcal{P}[\{e\}] & := (\{\mathcal{P}[e]\}, \bot) & \mathcal{P}[\bigcup\{e_2 \mid x^s \in e_1\}] & := \big(\bigcup\{\pi_1\, \mathcal{P}[e_2] \mid x^{\underline{s}} \in \pi_1 \mathcal{P}[e_1]\}, \bot\big)
\end{array}$$

$$\mathcal{P}[\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4] := \mathsf{if}\ val(\mathcal{P}[e_1]) = val(\mathcal{P}[e_2])\ \mathsf{then}\ \mathcal{P}[e_3]\ \mathsf{else}\ \mathcal{P}[e_4]$$

PROVENANCE SEMANTICS OF $\mathcal{NUL}$.

$$\begin{array}{rlrl}
\mathcal{P}[\mathsf{skip}^s] & := \mathsf{skip}^{\underline{s}} & \mathcal{P}[u; u'] & := \mathcal{P}[u]; \mathcal{P}[u'] \\
\mathcal{P}[\mathsf{repl}^s\ e] & := \mathsf{repl}^{\underline{s}}\ \mathcal{P}[e] & \mathcal{P}[[x^s]\, u] & := [x^{\underline{s}}]\, \mathcal{P}[u] \\
\mathcal{P}[\mathsf{insert}\ e] & := \mathsf{updl\ insert}\, (\pi_1\, \mathcal{P}[e]) & \mathcal{P}[\mathsf{iter}\ u] & := \mathsf{updl\ iter}\ \mathcal{P}[u] \\
\mathcal{P}[\mathsf{updl}\ u] & := \mathsf{updl\ updl}\ \mathcal{P}[u] & \mathcal{P}[\mathsf{updr}\ u] & := \mathsf{updl\ updr}\ \mathcal{P}[u]
\end{array}$$

$$\mathcal{P}[\mathsf{remove}\ e] := \mathsf{updl}\ \big([x]\ \mathsf{remove}\, \{y \mid y \in x, val(y) \in val(\mathcal{P}[e])\}\big)$$

Fig. 4. Provenance semantics of $\mathcal{NRA}$, $\mathcal{NRC}$, and $\mathcal{NUL}$.

Example 4. The provenance semantics $\mathcal{P}[\mathit{\Pi}_1]$ of the $\mathcal{NRA}$ expression $\mathit{\Pi}_1$ defining the left relational projection from Example 1 maps the colored set $R\colon \underline{\{b \times b\}}$ from Fig. 3 to the colored set R' in that figure. The provenance semantics of the $\mathcal{NRC}$ expression $\mathit{\Pi}_1$ defining the left relational projection from Example 2 has the same behavior. The provenance semantics of the $\mathcal{NRA}$ expression *cartprod* from Example 1 maps the colored pair $v\colon \underline{\{b\} \times \{b\}}$ from Fig. 5 to the colored set $w\colon \underline{\{b \times b\}}$ in that figure. The provenance semantics $\mathcal{P}[e_{(a)}]$ of the $\mathcal{NRC}$ expression $e_{(a)}$ from Example 2 that defines query (a) from the Introduction has already been illustrated: it maps the colored set R from Fig. 1 to the colored set in Fig. 1(a).

Note that expressions that are equivalent under the normal semantics need not be equivalent under the provenance semantics. For example, $map(id)$ is equivalent to id, but $\mathcal{P}[map(id)]$ is not equivalent to $\mathcal{P}[id]$ as the set returned by $\mathcal{P}[map(id)]$ is colored with $\bot$, while $\mathcal{P}[id]$ retains the original color from the input. Likewise, if x is a variable of type $s \times t$ then $(\pi_1\, x, \pi_2\, x)$ is equivalent to x, but $\mathcal{P}[(\pi_1\, x, \pi_2\, x)]$ is not equivalent to $\mathcal{P}[x]$.

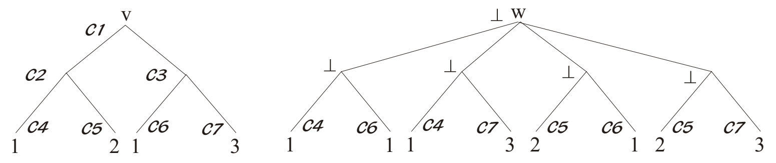

Fig. 5. The provenance semantics of cartesian product.

Define $\mathcal{PNRA}$ and $\mathcal{PNRC}$ as the languages we obtain by interpreting $\mathcal{NRA}$ and $\mathcal{NRC}$ under the new provenance semantics:

$$\mathcal{PNRA} := \{\ \mathcal{P}[f] \mid f \text{ expression in } \mathcal{NRA}\ \},$$
$$\mathcal{PNRC} := \{\ \mathcal{P}[e] \mid e \text{ expression in } \mathcal{NRC}\ \}.$$

Proposition 2. *$\mathcal{PNRA} \equiv \mathcal{PNRC}$ in the sense that every function definable by an expression $\mathcal{P}[f]\colon \underline{s} \to \underline{t}$ in $\mathcal{PNRA}$ is definable by a closed expression $\mathcal{P}[e]\colon \underline{s} \to \underline{t}$ in $\mathcal{PNRC}$, and vice versa.*

Hence, the equivalence of $\mathcal{NRA}$ and $\mathcal{NRC}$ (as stated by Proposition 1) continues to hold under the provenance semantics. In particular, we may continue to view expressions in $\mathcal{NRA}$ and $\mathcal{NRC}$ as "syntactic sugar" for expressions in the other language whenever convenient – even when we consider provenance. On the other hand, in order to study the expressive power of provenance in these languages it suffices to study the expressiveness of $\mathcal{PNRA}$ or $\mathcal{PNRC}$ alone. For example, the following is straightforward to prove by induction on f:

Proposition 3. *Every $\mathcal{PNRA}$ expression $\mathcal{P}[f]\colon \underline{s} \to \underline{t}$ defines a copying pado.*

It readily follows from Proposition 2 that every $\mathcal{PNRC}$ expression also defines a copying pado. The key result of this section is that the converse also holds:

Theorem 2. *Every function in $\mathcal{CP}$ is also definable by a closed expression $\mathcal{P}[f']\colon \underline{s} \to \underline{t}$ in $\mathcal{PNRC}$.*

This theorem essentially follows from the following observations. First, Theorem 1 continues to hold in the presence of the base type *color*. Hence, every pado in $\mathcal{CP}$ can be expressed by some closed expression $f\colon \underline{s} \to \underline{t}$ in $\mathcal{NRC}(color)$. Second, the color-propagation of f implies that f is "polymorphic on colors" in the sense that we can substitute the colors in f by objects of some other type as follows. Let r be an arbitrary type, let $g\colon r$ be a closed $\mathcal{NRC}$ expression, and let $\mathcal{T}[f, g]\colon s * r \to t * r$ be the $\mathcal{NRC}$ expression we obtain by replacing every occurrence of *color* in a type annotation in f by r and subsequently replacing every occurrence of the constant $\perp$ in f by g.

Example 5. Let $f\colon \underline{b \times \{b\}} \to \underline{b \times \{b\}}$ be as below. Then $\mathcal{T}[f, g]$ is as shown.

$$f = \lambda x^{\underline{b\times\{b\}}}.\Big(\big((5,\perp), (\pi_1\,\pi_2\,\pi_1\,x \cup \{\pi_1\,\pi_1\,x\}, \perp)\big), \perp\Big),$$
$$\mathcal{T}[f,g] = \lambda x^{(b\times\{b\})*r}.\Big(\big((5,g), (\pi_1\,\pi_2\,\pi_1\,x \cup \{\pi_1\,\pi_1\,x\}, g)\big), g\Big).$$

Note that $\mathcal{T}[f,g]$ propagates the objects of type r from input to output in the same way as f propagates colors, where g takes the role of $\bot$. What is more, $\mathcal{P}[\mathcal{T}[f,g]]\colon \underline{s * r} \to \underline{t * r}$ propagates the *colored* objects of type $\underline{r}$ from input to output in the same way as f propagates colors. The formal statement of this claim is as follows.

Let r and s be types and let $\phi\colon$ *color* $\to \underline{r}$ be a function. We define $w\colon \underline{s * r}$ to be *a substitution of the colors in* $v\colon \underline{s}$ *relative to* ϕ, denoted by $v \approx_s^\phi w$, by induction on s:

– $(a, c) \approx_b^\phi (((a, c'), \phi(c)), c'')$ with c' and c'' arbitrary;
– $((v, v'), c) \approx_{s \times s'}^\phi (((w, w'), \phi(c)), c')$ if $v \approx_s^\phi w$ and $v' \approx_{s'}^\phi w'$; and
– $(\{v, \dots, v'\}, c) \approx_{\{s\}}^\phi ((\{w, \dots, w'\}, \phi(c)), c')$ if $v \approx_s^\phi w, \dots, v' \approx_s^\phi w'$.

Proposition 4 (Color polymorphism). *Let $f\colon \underline{s} \to \underline{t}$ be a closed expression in $\mathcal{NRC}(color)$ defining a color-propagating function; let $g\colon r$ be a closed expression in $\mathcal{NRC}$; and let $\phi\colon$ color $\to \underline{r}$ be a function such that $\phi(\bot) = \mathcal{P}[g]$. Then $f(v) \approx_t^\phi \mathcal{P}[\mathcal{T}[f,g]]\,(w)$ for every $v\colon \underline{s}$ and every $w\colon \underline{s * r}$ with $v \approx_s^\phi w$.*

Let us now sketch how color polymorphism allows us to prove Theorem 2. In general, given a particular copying pado $f\colon \underline{s} \to \underline{t}$ in $\mathcal{NRC}(color)$ the proof constructs a type r and closed expressions $g\colon r$, $enc\colon s \to s{*}r$, and $dec\colon t{*}r \to t$ such that $\mathcal{P}[enc]\colon \underline{s} \to \underline{s * r}$ encodes the colors in $\underline{s}$ as colored objects of type $\underline{r}$ in $\underline{s * r}$ and $\mathcal{P}[dec]\colon \underline{t * r} \to \underline{t}$ decodes the colored objects of $\underline{r}$ in $\underline{t * r}$ back into their original colors. The copying property of f is crucial for the decoding step. Theorem 2 then follows as f is expressed in $\mathcal{PNRC}$ by $\mathcal{P}[dec \circ \mathcal{T}[f,g] \circ enc]$.

The construction. To illustrate the construction, assume that $f\colon \underline{b \times \{b\}} \to \underline{b \times \{b\}}$ is a copying pado in $\mathcal{NRC}(color)$. We will only motivate why f is equivalent to $\mathcal{P}[dec \circ \mathcal{T}[f,g] \circ enc]$ on *distinctly* colored objects. Equivalence on arbitrary colored object then follows by Lemma 1, as both f and $\mathcal{P}[dec \circ \mathcal{T}[f,g] \circ enc]$ are color-propagating. Furthermore, we will use tuples of arbitrary arity, as these can readily be simulated in the $\mathcal{NRC}$. For example, (x, y, z) is an abbreviation of $((x, y), z)$ and the projection π_i^n that retrieves the i-th component of an n-tuple is an abbreviation of $\pi_2\,\pi_1\,x$.

Let s abbreviate $b \times \{b\}$. Roughly speaking, we want $\mathcal{P}[enc]\colon \underline{s} \to \underline{s * r}$ to substitute every color in a distinctly colored object $v\colon \underline{s}$ by the unique colored subobject of v that is colored by c. In particular, $\underline{r}$ must hence be big enough to store all colored subobjects of v. Hereto, we take $r = b \times \{b\} \times (b \times \{b\}) \times \{unit\}$, where the first three components will be used to store colored subobjects from v, and the last type $\{unit\}$ will be used as an extra boolean flag that indicates the encoding of $\bot$. The following expressions can then be used to “inject” subobjects of v into r and to encode $\bot$:

$$\begin{array}{llll} put^b : b \to r := \lambda x^b.(x, e_{\{b\}}, e_s, \{\}) & put^{\{b\}} : \{b\} \to r := \lambda x^{\{b\}}.(e_b, x, e_s, \{\}) \\ put^s : s \to r := \lambda x^s.(e_b, e_{\{b\}}, x, \{\}) & g : r \quad := (e_b, e_{\{b\}}, e_s, \{()\}). \end{array}$$

Here, $e_b\colon b$, $e_{\{b\}}\colon \{b\}$, and $e_s\colon s$ are arbitrary but fixed closed $\mathcal{NRC}$ expressions. For example, e_s could be $(a, \{\})$ with $a\colon b$ an arbitrary constant.

We now construct enc such that if $v\colon \underline{s}$ is distinctly colored and $\phi\colon\ \mathit{color} \to \underline{r}$ is the function that maps $\bot$ to $\mathcal{P}[g]$ and that maps every color c occurring in v to $\mathcal{P}[\mathit{put}^{s'}](v_c)$ where $v_c\colon \underline{s'}$ is the unique subobject of v colored by c, then $v \approx_s^\phi \mathcal{P}[\mathit{enc}](v)$. Hereto, it suffices to let enc be

$$\lambda x^s.\, \Big(\big((\pi_1\, x, \mathit{put}^b(\pi_1\, x)), (\textstyle\bigcup\{\{(y, \mathit{put}^b(y))\} \mid y \in \pi_2\, x\}, \mathit{put}^{\{b\}}(\pi_2\, x))\big), \mathit{put}^s(x)\Big).$$

Let $w\colon \underline{s * r}$ abbreviate $\mathcal{P}[\mathcal{T}[f,g] \circ \mathit{enc}](v)$. Then $f(v) \approx_s^\phi w$ by color polymorphism since $v \approx_s^\phi \mathcal{P}[\mathit{enc}](v)$. That is, subobjects of type $\underline{r}$ in w are substitutions of the colors in $f(v)$ relative to ϕ. By inspecting these objects we can decode w back into $f(v)$ as follows. Let c be the color of $f(v)$, i.e., let $c = \pi_2(f(v))$. First, we note that we can check in $\mathcal{PNRC}$ whether $c = \bot$ in the sense that $\mathcal{P}[\lambda x.\mathsf{if}\ \pi_2\, x = g\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2](w)$ executes $\mathcal{P}[e_1]$ if $c = \bot$, and executes $\mathcal{P}[e_2]$ otherwise. To prove this claim, it suffices to show that $c = \bot$ iff $\mathit{val}(\mathcal{P}[\pi_2](w)) = \mathit{val}(\mathcal{P}[g])$. Suppose that $c = \bot$. Because $f(v) \approx_s^\phi w$, it is easily seen that $\mathcal{P}[\pi_2](w) = \phi(c) = \phi(\bot) = \mathcal{P}[g]$, and hence also $\mathit{val}(\mathcal{P}[\pi_2](w)) = \mathit{val}(\mathcal{P}[g])$. For the only-if direction, suppose for the purpose of contradiction that $\mathit{val}(\mathcal{P}[\pi_2](w)) = \mathit{val}(\mathcal{P}[g])$ but $c \neq \bot$. Since f is color-propagating, every color different from $\bot$ occurring in $f(v)$ must also occur in v. Hence, c occurs in v, and thus $\phi(c) = \mathcal{P}[\mathit{put}^{s'}](v_c)$. Because $f(v) \approx_s^\phi w$, it is easily seen that $\mathcal{P}[\pi_2](w) = \phi(c) = \mathcal{P}[\mathit{put}^{s'}](v_c)$. By construction, however, $\mathit{val}(\mathcal{P}[\mathit{put}^{s'}](v')) \neq \mathit{val}(\mathcal{P}[g])$ for any s' and any $v'\colon \underline{s'}$. Hence, $\mathit{val}(\mathcal{P}[\pi_2](w)) = \mathit{val}(\mathcal{P}[\mathit{put}^{s'}](v_c)) \neq \mathit{val}(\mathcal{P}[g])$, which gives us the desired contradiction.

We now claim that $\mathcal{P}[\mathit{dec}]$ with $\mathit{dec}\colon s * r \to s$ defined as follows successfully decodes w back into $f(v)$.

$$\begin{aligned}
\mathit{dec} &:= \lambda x.\ \mathsf{if}\ \pi_2\, x = g\ \mathsf{then}\ \big(\mathit{dec}^b(\pi_1\, \pi_1\, x), \mathit{dec}^{\{b\}}(\pi_2\, \pi_1\, x)\big)\ \mathsf{else}\ \pi_3^4(\pi_2\, x)\\
\mathit{dec}^b &:= \lambda x.\ \mathsf{if}\ \pi_2\, x = g\ \mathsf{then}\ \mathit{inv}(\pi_1\, x)\ \mathsf{else}\ \pi_1^4(\pi_2\, x)\\
\mathit{dec}^{\{b\}} &:= \lambda x.\ \mathsf{if}\ \pi_2\, x = g\ \mathsf{then}\ \textstyle\bigcup\{\{\mathit{dec}^b(y)\} \mid y \in \pi_1\, x\}\ \mathsf{else}\ \pi_2^4(\pi_2\, x).
\end{aligned}$$

Here, $\mathit{inv} := \lambda x.\,\mathsf{if}\ x = a_1\ \mathsf{then}\ a_1\ \mathsf{else}\ \ldots\ \mathsf{else\ if}\ x = a_k\ \mathsf{then}\ a_k\ \mathsf{else}\ x$, where $\{a_1, \ldots, a_k\}$ is the finite set of constants testifying that f is bounded inventing.

To see why $\mathcal{P}[\mathit{dec}](w) = f(v)$, first consider the case where $c \neq \bot$. Because f is copying, we know that $f(v) = v_c$ with $v_c\colon \underline{s}$ the unique subobject of v colored by c. Hence, $\mathcal{P}[\mathit{dec}](w) = \mathcal{P}[\pi_3^4](\mathcal{P}[\pi_2](w)) = \mathcal{P}[\pi_3^4](\phi(c)) = \mathcal{P}[\pi_3^4](\mathcal{P}[\mathit{put}^s](v_c))$. It is not hard to see that the latter is precisely $v_c = f(v)$, as desired.

Next, consider the case where $c = \bot$. Then $f(v)$ is a "newly constructed" colored pair. Hence, to decode w into $f(v)$, $\mathcal{P}[\mathit{dec}]$ first decodes $w_1 := \mathcal{P}[\pi_1\, \pi_1](w)$ and $w_2 := \mathcal{P}[\pi_2\, \pi_1](w)$ into $\pi_1(f(v))$ and $\pi_2(f(v))$ respectively, and constructs a new pair to put them in. Here, $\mathcal{P}[\mathit{dec}^b]$ and $\mathcal{P}[\mathit{dec}^{\{b\}}]$ decode w_1 and w_2 using essentially the same reasoning as $\mathcal{P}[\mathit{dec}]$: first they inspect the colors of w_1 and w_2, extracting the correct value from the $\underline{r}$-component if the color is not $\bot$, and by "reconstructing" the object otherwise.

This concludes the proof illustration of Theorem 2. As a corollary to Proposition 2, Proposition 3, and Theorem 2 we immediately obtain:

Corollary 1. *$\mathcal{PNRA}$, $\mathcal{PNRC}$, and $\mathcal{CP}$ are all equally expressive.*

5 Provenance for Updates

In this section we give an intuitive provenance-aware semantics for updates. Concretely, we take the view that updates do not construct new objects, but modify existing ones. As such, objects retain their colors during an update.

The provenance semantics $\mathcal{P}[u] \colon \underline{s} \to \underline{t}$ of a $\mathcal{NUL}$ update $u \colon s \to t$ is formally defined in Fig. 4 by translation into $\mathcal{NRL}(\mathit{color})$. Here, $\mathcal{P}[e]$ is the provenance semantics of $\mathcal{NRC}$ expression e as defined in Section 4 and $\{y \mid y \in x, \mathit{val}(y) \in \mathit{val}(\mathcal{P}[e])\}$ abbreviates the expression

$$\bigcup\{\mathsf{if}\ \mathit{val}(y) \in \mathit{val}(\mathcal{P}[e])\ \mathsf{then}\ \{y\}\ \mathsf{else}\ \{\} \mid y \in x\},$$

where the conditional $\mathsf{if}\ e_1 \in e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4$ is known to be expressible in $\mathcal{NRL}$ [7]. Note in particular that the provenance semantics of $\mathsf{remove}\ e$ ignores colors when selecting the objects to remove.

Example 6. The provenance semantics of the $\mathcal{NUL}$ update $\mathsf{iter}\ ([x]\ \mathsf{repl}\ \pi_1\ x)$ from Example 3 maps the colored set $R \colon \underline{\{b \times b\}}$ from Fig. 3 to the colored set R'' in that figure. Note in particular that the set itself retains its color. This is in contrast to the provenance semantics of the relational projection Π_1, as we have explained in Example 4. The provenance semantics of the updates $u_{(b)}$ and $u_{(c)}$ from Example 3 that express respectively the SQL updates (b) and (c) from the Introduction has already been illustrated in the Introduction. In particular, $\mathcal{P}[u_{(b)}]$ maps the colored set R from Fig. 1 to the colored set in Fig. 1(b), while $\mathcal{P}[u_{(c)}]$ maps R to the colored set in Fig. 1(c).

Define $\mathcal{PNUL}$ as the language we obtain by interpreting $\mathcal{NUL}$ under the new provenance semantics:

$$\mathcal{PNUL} := \{\mathcal{P}[u] \mid u \text{ expression in } \mathcal{NUL}\}.$$

It is easy to show that $\mathcal{PNUL} \subseteq \mathcal{KP}$; that is, every $\mathcal{P}[u]$ defines a kind-preserving pado. The key result of this section is that the converse also holds. The proof uses the same "color polymorphism" technique we have used for queries.

Theorem 3. *$\mathcal{KP} \equiv \mathcal{PNUL}$ in the sense that every function definable by an expression $f \colon \underline{s} \to \underline{t}$ in $\mathcal{KP}$ is also definable by a closed update $\mathcal{P}[u] \colon \underline{s} \to \underline{t}$ in $\mathcal{PNUL}$, and vice versa.*

6 Discussion

Our goal in this paper has been to achieve an understanding of how query and update languages manipulate provenance. Although the completeness results from Sections 4 and 5 show why our proposed provenance semantics is sensible, there are several issues that must be tackled before we can build a practical system that records provenance.

Space and processing overhead are concerns even for simple, manual updates [5]. From a space-efficiency point of view, it may be desirable to "merge" objects in the same set that differ. For example, we could collapse two edges in Figure 3 into a single edge labelled $\{c_5, c_7\}$. This is done for annotations in in [15, 3]. Query rewriting is also problematic. In Section 4 we noted that expressions that are equivalent under traditional semantics are no longer equivalent when provenance is considered. This may affect query optimisation.

There is also the issue of aggregation queries such as `select` A, `sum(`B`)` `from` R `group by` A. This particular aggregation could be expressed in $\mathcal{NRL}$ by adding a function $\mathit{sum}\colon \{\mathit{int}\} \to \mathit{int}$. Since the output of *sum* is a new data value, we could define $\mathcal{P}[\mathit{sum}] := \langle \mathit{sum}, K\bot \circ\, ! \rangle$, but it is surely more satisfactory to record some form of *workflow* provenance, as known from the geospatial and Grid computing communities [4, 9, 12], that tells us how the sum was formed. Another problem is that $\mathcal{P}[\mathit{sum}]$ is no longer bounded-inventing, a problem that also arises when we want to consider external user-defined functions. We hope to generalize our approach to address these issues.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations Of Databases.* Addison-Wesley, 1995.
2. O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: databases with uncertainty and lineage. In *VLDB 2006*, pages 953–964, 2006.
3. D. Bhagwat, L. Chiticariu, W. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *VLDB 2004*, pages 900–911, 2004.
4. R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.
5. P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD 2006*, pages 539–550, 2006.
6. P. Buneman, S. Khanna, and W. Tan. Why and where: A characterization of data provenance. In *ICDT 2001*, volume 1973 of *LNCS*, pages 316–330. Springer, 2001.
7. 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.
8. 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.
9. I. Foster and L. Moreau, editors. *Proceedings of the 2006 International Provenance and Annotation Workshop (IPAW 2006).* Number 4145. Springer-Verlag, 2006.
10. F. Geerts, A. Kementsietsidis, and D. Milano. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE 2006*, page 82, 2006.
11. H. Liefke and S. B. Davidson. Specifying updates in biomedical databases. In *SSDBM*, pages 44–53, 1999.
12. Y. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.
13. W. Tan. Containment of relational queries with annotation propagation. In *DBPL 2003*, volume 2921 of *LNCS*, pages 37–53. Springer, 2003.
14. W. Tan. Research problems in data provenance. *IEEE Data Engineering Bulletin*, 27(4):45–52, 2004.
15. Y. R. Wang and S. E. Madnick. A polygen model for heterogeneous database systems: The source tagging perspective. In *VLDB 1990*, pages 519–538, 1990.