A Relational Algebra for Basic Terms in a Higher-Order Logic

Simon Price and Peter A. Flach
Department of Computer Science, University of Bristol, Bristol BS8 1UB, UK
simon.price@bristol.ac.uk

Technical Report CSTR-13-004
July 2013

Abstract

We define a relational algebra on basic terms, strongly typed terms in a higher-order logic, that are well suited to the representation of heterogeneous data, irrespective of whether the data originated from relational, unstructured, semi-structured or structured sources. This higher-order generalisation of the relational model has potential applications in NoSQL databases and Big Variety, Big Data applications.

Contents

1 Introduction

The relational model is the *de facto* standard for database-driven applications but is not ideally suited for representing semi-structured data such as Web pages, XML and numerous other annotated textual document formats. Neither is the relational model convenient for representing structured data such as trees, lists, bags and so on, although the representation of such structures in relational databases is commonplace using a multitude of (often tortuous) representations and querying patterns [6, 7, 9, 13].

The individuals-as-terms representation is a generalisation of the relational model's attribute-value representation and collects all information about an individual in a single term. As such, the individuals-as-terms representation has much in common with the various hierarchical and object database models that were the forerunners of the highly successful relational model that all but replaced them – until the recent Big Data driven emergence of NoSQL databases caused a resurgence of interest in non-relational data models.

In contrast to the relational model, the individuals-as-terms model offers straight forward representations of both structured and semi-structured data while at the same time having the representational capacity to represent relations from the relational model. This representational flexibility makes the individuals-as-terms model a convenient representation of heterogeneous data, enabling the collection of all information about an individual in a single term irrespective of whether that information is relational, semi-structured or structured.

This paper is motivated by a practical requirement to not only represent heterogeneous data as terms but also to be able to query and merge those terms with all the convenience of the relational algebra. The terms in question are the *basic terms* of a strongly-typed, higher-order logic [11]. We show that the relational model can be lifted to a more general, higher-order relational model that has promising applications for querying and merging heterogeneous data [14].

The remainder of this paper is organised as follows: in Section 2 we review the traditional relational model, presenting the relational algebra and its operations in a form that neatly maps to our higher-order generalisations; in Section 3 we introduce the higher-order representation and various methods of referring to sub-parts of basic terms; in Section 4 we lift the relational algebra and its operators to our higher-order relational model; Section 5 gives a brief summary and suggests future work.

2 Relational Model

The relational model for database management was first introduced by Edgar Codd at IBM Research [2, 3] and was subsequently refined by him and others over the following two decades [4, 5]. The relational model provides a precise specification for what a relational database should do but, deliberately, does not specify how such a relational database should be built. Most of today's database management systems, including all those which employ variants of the SQL query language, are based on ideas drawn from the relational model. Underpinning the relational model is a principled theoretical foundation that draws on the mathematical topics of set theory, first-order predicate logic, and the theory of types. For the purposes of this paper, the most important aspects of the relational model are the relational representation along with its associated operations and algebra, which we recall below.

2.1 Relational Representation

Surveying the relational database literature can initially be confusing because authors adopt one of two alternative definitions of the relational representation. Indeed Codd himself switched from one representation to the other early on in his development of the relational model. The competing representations differ in their definition of the tuple: the earlier representation follows the conventional mathematical definition of a tuple whereas the later representation defines the, so called, *unordered labelled tuple*. In this paper we follow Codd's original relational representation for reasons that we discuss below.

An n-tuple $(x_1, \ldots, x_n)$ is an element in the Cartesian product $D_1 \times \cdots \times D_n$ where $x_1, \ldots, x_n$ are values drawn from domains $D_1, \ldots, D_n$ respectively, $n \in \mathbb{N}$ and $n \geq 1$. We refer to n-tuples as *tuples* unless the value of n is significant. The core of the relational representation is the *relation*, which is a homogeneous set of such tuples defined as follows.

Definition 2.1 (Relation) *A* relation R *of degree* n *is a finite set of* n*-tuples such that* $R \subseteq D_1 \times \cdots \times D_n$ *where* $D_1, \ldots, D_n$ *are domains.*

The *schema* (or *scheme*) of a relation $R \subseteq D_1 \times \cdots \times D_n$ is denoted $R(D_1, \ldots, D_n)$ and R is said to be a relation on that schema. The domains $D_1, \ldots, D_n$ in a relation need not necessarily be distinct. All domain values are considered to be atomic, meaning that they are indivisible as far as the relational model is concerned. Also a special value called `NULL` is included in every domain[1].

[1] The multiple roles of `NULL` values in the relational model fall outside the scope of this paper but a comprehensive discussion appears in [7].

Being a set, duplicate tuples are not permitted in a relation. This is a notable difference between the relational model and typical implementations of relational databases, where duplicate tuples are permitted in relations because a multiset (or bag) representation is used instead of a set representation[2]. In the relational database literature a relation is often referred to as a table, and a tuple as a record or row in a table. By definition, the rows of a table occur in a specific order whereas the elements of a set are entirely unordered and so a table is a rather inaccurate representation of a relation.

Prior to defining the relational operations and algebra it is first necessary to define some means for identifying values at specific positions within a tuple. Item $i \in \{1, \ldots, n\}$ of a tuple $t = (x_1, \ldots, x_n)$ is the value x_i and is referred to as $t|_i$. The set of tuple item indices for a given relation is the *relation index* of the relation.

Definition 2.2 (Relation Index) *The* relation index I_R *of a relation* R *of degree* n *is the set* $\{1, \ldots, n\}$.

The relation index, and other indexes to be described, in this paper should not be confused with the sorts of indexes used to increase record access speed within relational database implementations.

The relation index I_R for some relation R is isomorphic with the set of attribute names $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ in the traditional *name-based schema* $\mathcal{R}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$, where $\mathcal{R}$ is the relation name associated with R, and $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are attribute names associated with domains $D_1, \ldots, D_n$ respectively. For example, if `author(firstname, lastname, email)` is the name-based schema for a relation R representing authors in a bibliographic database then the set of attribute names $\{\texttt{firstname}, \texttt{lastname}, \texttt{email}\}$ is trivially isomorphic with the relation index $I_R = \{1, 2, 3\}$. Hence, without loss of generality, we use the relation index instead of the name-based schema in order to simplify the upgrading of the relational model to higher-order logic.

In keeping with our choice of the relation index, we also adopt Codd's original definition of the tuple for the elements of the relation as opposed to his subsequent alternative of the unordered labelled tuple. The latter relaxes the definition of a tuple so that the order of its items becomes irrelevant. More formally, an unordered labelled tuple u is defined as a set of name-value pairs such that $u = \{(\mathcal{A}_1, x_1), \ldots, (\mathcal{A}_n, x_n)\}$, where each $\mathcal{A}_i$ and x_i pair is a corresponding attribute name and item value. Pairing each value with an identifying attribute name brings the convenience of set operations on tuple items in defining the relational operations. Associativity and commutativity of various relational operations is achieved as a consequence but at the cost of the complexity of maintaining the attribute name uniqueness constraint, $\forall i, j \in 1 \ldots n, \mathcal{A}_i = \mathcal{A}_j \implies i = j$, which in turn gives rise to a requirement for a *rename* operation [13]. In our higher-order setting, as we later explain, operation associativity and commutativity cease to be relevant and so we are able to avoid this unnecessary complexity through our choice of tuple.

2.2 Relational Operations

The relational algebra defines eight operations on relations. Five of these operations (union, difference, product, projection and restriction) are considered fundamental, or primitive, as they can not be derived from combinations of the other relational operations. The remaining three (intersection, divide and join) can each be derived by combining the fundamental operations. The following sections define all eight of operations, beginning with the fundamental ones. Practical implementations of the relational algebra also include additional convenience operations, aggregate operations and update operations that are not strictly part of the algebra and which we do not consider here.

2.2.1 Fundamental Relational Operations

The relational algebra defines the following five operations.

- relational union ($\cup$)
- relational difference ($-$)
- relational product ($\times$)
- relational projection (π)
- relational restriction (σ)

[2] Relational databases typically follow SQL standards and consequently deviate from the relational model by allowing duplicate elements in relations, thereby invalidating the model's theoretical results [5].

The definitions of relational union and difference (and, later when we discuss the non-fundamental operations, intersection) accord with the usual definition of the corresponding operations from set theory but apply solely to sets that are relations *union compatible*. Relations A and B are union compatible if and only if $A, B \subseteq D_1 \times \cdots \times D_n$. As would be expected from the normal definition of a set, all duplicate tuples are removed from the result (co-domain) of any of the following operations.

Definition 2.3 (Relational Union) *The* relational union $A \cup B$ *of relations* $A, B \subseteq D_1 \times \cdots \times D_n$ *is the relation* $A \cup B = \{t \mid t \in A \vee t \in B\}$.

Definition 2.4 (Relational Difference) *The* relational difference $A - B$ *of relations* $A, B \subseteq D_1 \times \cdots \times D_n$ *is the relation* $A - B = \{t \mid t \in A \wedge t \notin B\}$.

The definition of the relational product operation differs from the normal Cartesian product in that the result is a set of single tuples rather than a set of pairs of tuples. The single tuple is formed by applying a tuple concatenation function to the pair of tuples that would result from a conventional Cartesian product. *Tuple concatenation* is a binary function, *conc*, on tuples such that for tuples $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_m)$, $conc(a, b) = (a_1, \ldots, a_n, b_1, \ldots, b_m)$. The tuples a and b need not be union compatible.

Definition 2.5 (Relational Product) *The relational product* $A \times B$ *of relations* A *and* B *is* $A \times B = \{conc(a, b) \mid a \in A, b \in B\}$.

One consequence of our definition of relations is that the relational product is neither associative nor commutative. There does however exist a bijection between the tuple items of the different concatenated tuples. So $A \times B \equiv B \times A$ up to isomorphism. However, we will address the lack of associativity and commutativity later in this paper.

Definition 2.6 (Relational Projection) *Let* A *be a relation with corresponding relation index* I_A. *Let* ρ *be a list of relation index items drawn from* I_A *such that* $\rho = [i_1, \ldots, i_n]$. *The* relational projection π *on* ρ *of* A *is:* $\pi_\rho(A) = \{(a|_{i_1}, \ldots, a|_{i_n}) \mid a \in A\}$.

Relational projection corresponds quite closely to `SELECT DISTINCT` in SQL.

Definition 2.7 (θ-Restriction) *Let* θ *be a predicate* $\theta : D \times D \to \mathbb{B}$ *for some domain* D. *For a relation* A, *with corresponding relation index* I_A, *the* θ-restriction σ_φ *of* A *is defined as follows.*

1. *If* φ *has the form* $i\,\theta j$ *where,* $i, j \in I_A$ *and* $a|_i, a|_j \in D$, *then*
$$\sigma_{i\,\theta j}(A) = \{a \mid a \in A \ \wedge \ a|_i\,\theta\,a|_j \ \}.$$

2. *If* φ *has the form* $i\,\theta(v)$, *where* $i \in I_A$ *and* $a|_i, v \in D$, *then*
$$\sigma_{i\,\theta(v)}(A) = \{a \mid a \in A \ \wedge \ a|_i\,\theta\,v \ \}.$$

The infix θ in the subscript of σ follows the historical convention from the relation database literature and so $i\,\theta j$, or equivalently $\theta(i, j)$, does not mean that θ applies to i and j; instead $i\,\theta j$ is shorthand notation for the membership test $a|_i\,\theta\,a|_j$ for all $a \in A$. Also, the parenthesised subscript (v) is our notation to distinguish v as a literal value as distinct to the index of a tuple item, such as i, where no parentheses are used.

The predicate θ is typically drawn from the set $\{=, \neq, <, \leq, >, \geq\}$ but does not necessarily have to come from this set. θ-restriction is often just referred to as *restriction* and in such cases θ is assumed to be the equality operation. The name *selection* is often used instead of *restriction* in the literature but we will not use this synonym here to avoid confusion with the `select` operation from SQL which has a somewhat different meaning. In fact, restriction corresponds more closely to the `WHERE` clause in SQL.

Definition 2.8 (Generalised Restriction) *Let* φ *be a proposition that consists of atoms as allowed in* θ*-restriction and the logical operations* $\wedge$, $\vee$ *and* $\neg$. *If* A *is a relation then the* generalised restriction σ_φ *is defined on* A *as follows:*
$$\sigma_\varphi(A) = \{\ a \mid a \in A \ \wedge \ \varphi(a)\ \}.$$

Proposition 2.9 *Generalised restriction does not increase the expressive power of the relational algebra if the algebra already includes θ-restriction.*

Proof 2.10 *Let φ and ψ be propositions on elements of relation A and let them consist only of atoms as allowed in θ-restriction. If we assume the usual set intersection, union and difference operations then the result follows directly from the equivalences: $\sigma_{\varphi\wedge\psi}(A) = \sigma_\varphi(A) \cap \sigma_\psi(A)$, $\sigma_{\varphi\vee\psi}(A) = \sigma_\varphi(A) \cup \sigma_\psi(A)$ and $\sigma_{\neg\varphi}(A) = A - \sigma_\varphi(A)$.*

2.2.2 Derivative Relational Operations

Definition 2.11 (Relational Intersection) *The* relational intersection $A \cap B$ *of relations $A, B \subseteq D_1 \times \cdots \times D_n$ is the relation $A \cap B = \{t \mid a \in A \wedge t \in B\}$.*

Expressed in terms of the fundamental relational operations $A \cap B = A - (A - B)$, $A \cap B = B - (B - A)$ and $A \cap B = A \cup B - (A - B) - (B - A)$.

Relational division is the reverse of the relational product.

Definition 2.12 (Relational Division) *The* relational division $A \div B$ *of relation $A \subseteq D_1 \times \cdots \times D_n \times D_{n+1} \times \cdots \times D_m$ and relation $B \subseteq D_{n+1} \times \ldots D_m$, where $n \leq m$, is the n-ary relation*

$$A \div B = \{\, t \mid \forall b \in B, \mathit{conc}(t, b) \in A\}.$$

Expressed in terms of the fundamental relational operations,

$$A \div B = \pi_{1,\ldots,n}(A) - \pi_{1,\ldots,n}((\pi_{1,\ldots,n}(A) \times B) - A).$$

Definition 2.13 (θ-Join) *Let θ be a predicate $\theta : D \times D \to \mathbb{B}$ for some domain D. If A and B are relations with tuple items $a|_i \in D$ and $b|_j \in D$ respectively for some $(i, j) \in I_A \times I_B$, then the θ-*join $\bowtie_{i\,\theta j}$ *of A and B is defined as*

$$A \bowtie_{i\,\theta j} B = \sigma_{i\,\theta j}(A \times B).$$

When θ is equality the θ-join is called the *equi-join*. By replacing the θ-restriction operation in the θ-join by the generalised restriction operation we arrive at the definition of the *generalised join*.

Definition 2.14 (Generalised Join) *Let φ be a proposition that consists of atoms as allowed in θ-restriction and the logical operations $\wedge$, $\vee$ and $\neg$. If A and B are relations then the* relational join $\bowtie_\varphi$ *is defined as*

$$A \bowtie_\varphi B = \sigma_\varphi(A \times B).$$

For the purposes of upgrading relational joins to handle structured data, it is sufficient to consider just the θ-join and, optionally as a useful syntactic convenience, the generalised join. However, a description of relational joins would not be complete without mentioning the, so called, *natural join*. The natural join is the result of a projection of an equi-join such that duplicate tuple items are removed from the resulting relation.

Definition 2.15 (Natural Join) *Let A and B be relations with tuples $a \in A$ and $b \in B$. If tuple items $a|_i \in D$ and $b|_j \in D$ for some domain D and some $(i, j) \in I_A \times I_B$ then the* natural join $\bowtie_{(i,j)}$ *of A and B is*

$$A \bowtie_{(i,j)} B = \pi_\rho(\sigma_{i=j}(A \times B)),$$

where $\rho = I_A \cup \{\, |I_A| + \ell \mid \ell \in I_B \wedge \ell \neq j\,\}$.

When unqualified reference is made in the literature to a "*relational join*", this typically refers to the *natural join*. The literature also defines several other types of join, including *semijoin*, *antijoin*, *outer joins* and *inner joins*. These may all be expressed in terms of the fundamental operations described in this paper but are not discussed further here and the interested reader is instead referred to standard texts such as [5, 6]. For the purposes of upgrading relational joins to handle structured data, it is sufficient to consider just the θ-join and optionally, as a useful syntactic convenience, the generalised join.

Natural joins as traditionally defined in Codd's relational algebra [3, 4] are both commutative, i.e. $A \bowtie B = B \bowtie A$, and associative, i.e. $(A \bowtie B) \bowtie C = A \bowtie (B \bowtie C)$. Neither of these properties holds for the relational joins described in this paper. In the context of traditional database applications, both of these properties are of practical importance. By contrast, in the context of joining structured data rather than relational data, the resultant structured data would not typically be expected to have the same structure as the original data and thus commutativity and associativity become less relevant.

For an exact relational join, the behaviour of the natural join in removing duplicate tuple items after joining on them is appropriate because the presence of duplicates adds no information to the data. However, in the context of an approximate relational join, the joined-on tuple items need not be identical – approximate equality is sufficient – and so both values may hold useful information. For our intended machine learning and data mining application domain, we choose not to remove this information.

2.3 Relational Algebra

Given the relational representation and fundamental operations defined above we can now define the *relational algebra* along similar lines to [13] but with alterations to accommodate our choice of representation.

Definition 2.16 (Relational Algebra) *Let $\mathcal{D}$ be a set of domains. Let dom be a total function from values $v \in D$ to their associated domain $D \in \mathcal{D}$ such that $dom(v) = D$. Let $\boldsymbol{R}$ be a set of relations such that $\boldsymbol{R} = \{R \mid R \subseteq D_1 \times \cdots \times D_n \ \wedge \ D_i \in \mathcal{D}, \ i = 1..n\}$. Let ind be a total function from relations $R \in \boldsymbol{R}$ to relation indexes $I_R \subseteq \mathbb{N}^+$. Let Θ be a set of predicates over domains in $\mathcal{D}$, and the logical operations $\wedge$, $\vee$ and $\neg$ over the booleans. Let $\boldsymbol{O}$ be the set of operations: relational union $\cup$, relational difference $-$, relational product $\times$, relational projection π, and relational restriction σ. The* relational algebra *over $\mathcal{D}$, dom, $\boldsymbol{R}$, ind, Θ and $\boldsymbol{O}$ is the 6-tuple $\mathscr{R} = (\mathcal{D}, dom, \boldsymbol{R}, ind, \Theta, \boldsymbol{O})$. An* algebraic expression *over $\mathscr{R}$ is any expression formed legally, according to the definitions of the operations, from the relations in $\boldsymbol{R}$ using the operations in $\boldsymbol{O}$.*

The derivative operations, such as relational intersection ($\cap$), θ-join ($\bowtie_\theta$), relational division ($\div$) and the natural join ($\bowtie$), are not defined as members of $\boldsymbol{O}$ because they may be defined from combinations of the fundamental operations.

Index set I_R of each relation $R \in \boldsymbol{R}$ does not appear in the definition directly, as would be the case with name-based relational schema, but is instead obtained through the function *ind* so that an association between relation and index is maintained.

The function *dom* ensures that every value in every domain in $\mathcal{D}$ is associated with its domain. This association between a value and its domain is an important feature in the upgrading of relational algebra to the higher-order logic where we exploit a similar relationship between a value and its type (to be defined).

3 Upgrading the Relational Model to Higher-Order Logic

3.1 Individuals as Terms in a Higher Order Logic

The setting is a typed, higher-order logic which is based on Church's simple theory of types [1] with several extensions. This formalism has been chosen over the possible alternative of first-order logic because terms in the higher-order logic natively support a variety of data types that are important for representing individuals, including sets, multisets and graphs. Furthermore, selecting a typed logic enables type inference on heterogeneous data about an individual, which simplifies the integrated representation in a term. The theory behind the logic and the individuals-as-terms formalism is set out in [11, 12] and a brief overview is given here. First, we assume an *alphabet* defined as follows.

Definition 3.1 (Alphabet [12]) *An* alphabet *consists of four sets.*

1. *$\mathfrak{T}$ the set of type constructors of various arities.*

2. *$\mathfrak{P}$ the set of parameters.*

3. *$\mathfrak{C}$ the set of constants.*

4. *$\mathfrak{V}$ the set of variables.*

Included in $\mathfrak{T}$ is the constructor Ω of arity 0 with a corresponding domain of $\{\mathit{True}, \mathit{False}\}$, the booleans. Types are constructed from type constructors in $\mathfrak{T}$ and type variables in $\mathfrak{P}$ using the symbols $\rightarrow$ for function types and $\times$ for product types.

Definition 3.2 (Type [12]) *A* type *is defined inductively as follows.*

1. *Each parameter in $\mathfrak{P}$ is a type.*
2. *If T is a type constructor in $\mathfrak{T}$ of arity k and $\alpha_1, \ldots, \alpha_k$ are types, then $T\ \alpha_1 \ldots \alpha_k$ is a type. (For $k = 0$, this reduces to a type constructor of arity 0 being a type.)*
3. *If α and β are types, then $\alpha \to \beta$ is a type.*
4. *If $\alpha_1, \ldots, \alpha_n$ are types, then $\alpha_1 \times \cdots \times \alpha_n$ is a type. (For $n = 0$, this reduces to 1 being a type.)*

A type is *closed* if it contains no parameters. $\mathfrak{S}^c$ denotes the set of all closed types obtained from an alphabet. We admit the usual nullary type constructors, including:

- 1,
- Ω, the type of $\mathbb{B}$,
- *Nat*, the type of $\mathbb{N}$,
- *Int*, the type of $\mathbb{Z}$,
- *Float*, the type of floating-point numbers,
- *Real*, the type of $\mathbb{R}$,
- *Char*, the type of characters,
- *String*, the type of strings.

The set of constants $\mathfrak{C}$ includes $\top$ (true) and $\perp$ (false). A *signature* is the declared type for a constant. A constant C with signature α is often denoted $C : \alpha$. Let $[\,]$ be the empty list constructor with signature *List* a where a is a parameter and *List* is a type constructor. Let $\#$ be the list constructor with signature $a \to$ *List* $a \to$ *List* a.

The *terms* of the logic are the terms of typed λ-calculus and are formed in the usual way by abstraction, tupling and application from constants in $\mathfrak{C}$ and a set of variables. The set of all terms obtained from a particular alphabet is denoted $\mathfrak{L}$ and is called the *language* given by the alphabet. A basic term is the canonical representative of an equivalence class of terms [8, 12].

Definition 3.3 (Basic terms [12]) *The set of* basic terms, *$\mathfrak{B}$, is defined inductively as follows.*

1. Basic structures – *If C is a data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_i \ldots a_k)$, $t_1, \ldots, t_n \in \mathfrak{B}$ $(n \geq 0)$, and t is $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $t \in \mathfrak{B}$.*
2. Basic abstractions – *If $t_1, \ldots, t_n \in \mathfrak{B}$, $s_1, \ldots, s_n \in \mathfrak{B}$ $(n \geq 0)$, $s_0 \in \mathfrak{D}$ and t is $\lambda x.$ if $x = t_1$ then s_1 else $\ldots$ if $x = t_n$ then s_n else $s_0 \in \mathfrak{L}$, then $t \in \mathfrak{B}$.*
3. Basic tuples – *If $t_1, \ldots, t_n \in \mathfrak{B}$ $(n \geq 0)$ and t is $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $t \in \mathfrak{B}$.*

The abstractions in part 2 of Definition 3.3 represent a key-value lookup table where t_i are the keys and s_i are the values, for $i = 1 \ldots n$, with a default value of $s_0 \in \mathfrak{D}$, where $\mathfrak{D}$ is the set of terms that do not already occur in $\{t_1, \ldots, t_n\}$. The order of the terms t_i and s_i in basic abstractions always in a canonical lexical total ordering to avoid syntactically different terms that are semantically equivalent.

Basic terms can represent a wide range of data structures using basic structures, basic abstractions and basic tuples, or arbitrarily nested combinations of these. We give a few important examples of data structures as basic terms below.

Let M be a nullary type constructor and $A, B, C, D : M$. Let $[\,]$ be the empty list constructor with signature *List* a where a is a parameter and *List* is a type constructor. Let $\#$ be the list constructor with signature $a \times$ *List* $a \to$ *List* a. In Fig. 5, the lists $[A, B, C]$ and $[A, D]$ are represented as right-descending binary trees, using nested basic structures.

Basic abstractions are key-value associations of type $\mathfrak{B}_{\beta \to \gamma}$, for some key of type β and value of type γ, with a default value $s_0 : \gamma$. The value t_i associated with key s_i for some basic abstractions t is $V(t\ s_i) = t_i$ for $i = 1, \ldots, n$. The set of keys s_i that occur in t is supp(t). With suitable choices of γ and s_0, basic abstractions

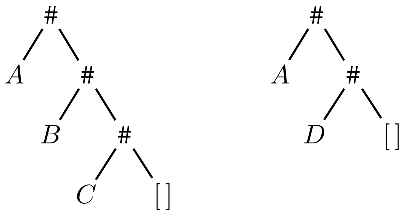


Figure 1: Lists $[A, B, C]$ and $[A, D]$ as basic structures.

can represent sets and multisets (bags). A set of terms $\{A, B, C\}$ of type M can be represented extensionally as basic abstractions of type $\mathfrak{B}_{M \to \Omega}$, where Ω is the type of booleans, and the default term $s_0 = \bot$.

$$\lambda x \,.\, \text{if } x = A \text{ then } \top \text{ else if } x = B \text{ then } \top \text{ else if } x = C \text{ then } \top \text{ else } \bot$$

A multiset $\langle A, A, A, B, C, C\rangle = \langle (A, 3), (B, 1), (C, 2)\rangle$ can be represented as basic abstractions of type $\mathfrak{B}_{M \to \mathit{Nat}}$, where *Nat* is the type of natural numbers and the default term $s_0 = 0$.

$$\lambda x \,.\, \text{if } x = A \text{ then } 3 \text{ else if } x = B \text{ then } 1 \text{ else if } x = C \text{ then } 2 \text{ else } 0$$

When working with terms and basic terms it is useful or, as in the context of the relational algebra on basic terms, necessary to be able to refer to sub-parts of a term either individually or collectively. We refer to the process of specifying a specific sub-part or set of sub-parts as *indexing*.

3.2 Indexing Basic Terms

In the logic, sub-parts of a term are referred to as subterms and so we are concerned with indexing the subterms of a basic term. The standard method for indexing subterms in the logic enumerates a decomposition of a given term such that every subterm is labelled with a unique string [12]. However, we introduce an alternative approach to indexing that, instead of enumerating all subterms of a term, defines a *type tree index set* over all subtypes of the type of a basic term. To do this we first adopt the definition of a type tree from [10] and then define a different annotation of the tree such that every member of the type tree index set identifies a set of terms rather than a single term. This ensures any index defined on a type is meaningful across all terms of that type. Furthermore, the set of subterms identified is guaranteed to consist entirely of well-formed basic terms.

Below we consider two subterm indexing methods which we call *term-based indexing* and *type-based indexing*, the latter of which has a variant which we also discuss.

3.2.1 Term-based Indexing

We refer to *term-based indexing* as the approach taken in [12], whereby the *occurrence set* $\mathcal{O}(t)$ of a term t is defined to enable subterm indexing such that the subterm of t at occurrence $o \in \mathcal{O}(t)$ may be referenced as $t|_o$. Each occurrence is either a numeric string that uniquely labels a subterm of a given term, or is the empty string ε, labelling the reflexive subterm. Example 3.4, using the trivial case of a basic tuple, informally illustrates subterm indexing based on a term's occurrence set as defined in [12].

Example 3.4 *If basic term t is the tuple $t = (A, B, C, D)$ such that $A, B, C, D \in \mathfrak{B}$, then the occurrence set of t is $\mathcal{O}(t) = \{\varepsilon, 1, 2, 3, 4\}$, and the subterms of t are indexed as $t|_\varepsilon = (A, B, C, D)$, $t|_1 = A$, $t|_2 = B$, $t|_3 = C$ and $t|_4 = D$.*

For the case of basic tuples, such a scheme is a suitable analogue of the relation index, because indexing over the items of basic tuples corresponds closely with numeric indexing over the items of tuples in the traditional relation index. If, however, we move beyond indexing basic tuples and onto basic structures or basic abstractions, then indexing directly on the terms themselves has two undesirable consequences, in our context, which we outline below.

Firstly, indexes over basic structures and basic abstractions requires foreknowledge of the extension of data instances in order to select a meaningful occurrence that applies to all the basic terms in a basic term relation, as illustrated for basic structures in Example 3.5. In other words, some occurrence values are local to a subset of the basic terms in the relation: possibly even local to a single term.

Example 3.5 *If basic terms $s, t \in \mathfrak{B}_{\mathit{List}\,M}$ are the lists $s = [A, B, C]$ and $t = [A, D]$, where $A, B, C, D : M$, and M is a nullary type constructor, then the occurrence sets of s and t are $\mathcal{O}(s) = \{\varepsilon, 1, 2, 21, 22, 221, 222\}$ and $\mathcal{O}(t) = \{\varepsilon, 1, 2, 21, 22\}$, the derivation of which can be seen from Figure2. The occurrence sets correspond to the subterms $s|_\varepsilon = [A, B, C]$, $s|_1 = A$, $s|_2 = [B, C]$, $s|_{21} = B$, $s|_{22} = [C]$, $s|_{221} = C$, $s|_{222} = []$, and $t|_\varepsilon = [A, D]$, $t|_1 = A$, $t|_2 = [D]$, $t|_{21} = D$, $t|_{22} = []$. So, for instance, occurrence 21 can be used to index both s and t because $21 \in \mathcal{O}(s) \cap \mathcal{O}(t)$. In contrast, occurrence 221 can only be used to index s because $221 \in \mathcal{O}(s)$, but $221 \notin (\mathcal{O}(s) \cap \mathcal{O}(t))$. Thus occurrence 221 is local to s with respect to t.*

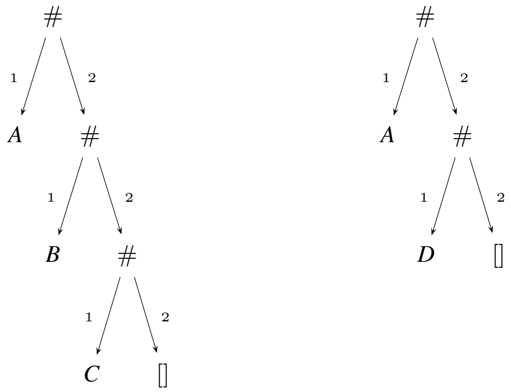


Figure 2: Term-based indexing for basic structures representing lists $[A, B, C]$ and $[A, D]$, which is notational sugar for $A\#B\#C\#[]$ and $A\#D\#[]$, where $\#$ and $[]$ are the usual list data constructors, $A, B, C, D : M$, and M is a nullary type constructor.

This *indexing locality problem* increases as the complexity and nesting of basic terms increases. Notably for abstractions, representing sets and multisets etc., the locality of occurrences is total; an occurrence has little or no meaning as an index outside of the abstraction term itself, as can be seen from Example 3.6.

Example 3.6 *Let basic terms $s, t \in \mathfrak{B}_{\alpha \to \Omega}$ be the sets $s = \{A, B, C\}$ and $t = \{B, D\}$, represented by the basic abstractions*

$$\begin{aligned} s =& \lambda x.\ \mathit{if}\ x = A\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = B\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = C\ \mathit{then}\ \top\ \mathit{else}\ \bot, \text{ and} \\ t =& \lambda x.\ \mathit{if}\ x = B\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = D\ \mathit{then}\ \top\ \mathit{else}\ \bot, \end{aligned}$$

where $A, B, C, D : M$, and M is a nullary type constructor. Given that "if p then q else r" is infix notation for if_then_else(p, q, r), these basic abstractions may also be rewritten as follows, additionally shortening "if_then_else" to "if" for conciseness:

$$\begin{aligned} s =& \lambda(x, \mathit{if}(=(x, A), \top, \mathit{if}(=(x, B), \top, \mathit{if}(=(x, C), \top, \bot)))), \text{ and} \\ t =& \lambda(x, \mathit{if}(=(x, B), \top, \mathit{if}(=(x, D), \top, \bot))). \end{aligned}$$

Their occurrence sets are $\mathcal{O}(s) = \{\varepsilon, 1, 2, 21, 22, 23, 211, 212, 231, 232, 233, 2331, 2332, 2333, 23311, 23312\}$ and $\mathcal{O}(t) = \{\varepsilon, 1, 2, 21, 22, 23, 211, 212, 231, 232, 233, 2311, 2312\}$, the derivation of which can be seen from Figure3. Out of these occurrences, the only ones likely to be of practical use in a join operation correspond to the subterms $s|_{212} = A$, $s|_{2312} = B$, $s|_{23312} = C$ and $t|_{212} = B$, $t|_{2312} = D$. These occurrences are clearly local to the term upon which they are defined.

The second undesirable consequence, in our context, of indexing directly on the terms using the occurrence set defined in [12] is that subterms are not guaranteed to be basic terms. For example, for any basic abstraction the variable x from $\lambda.x$ is a subterm but is not a basic term. In the context of exact relational joins, and leaving aside issues of redundancy or the semantics of joining on such subterms, this is not of itself problematic because the logic has a well defined equality operation over all terms, and all subterms are terms. However, for approximate joins, admitting subterms that are not basic terms prevents the application of existing approximate equality operations defined only for the subset of terms that are basic terms.

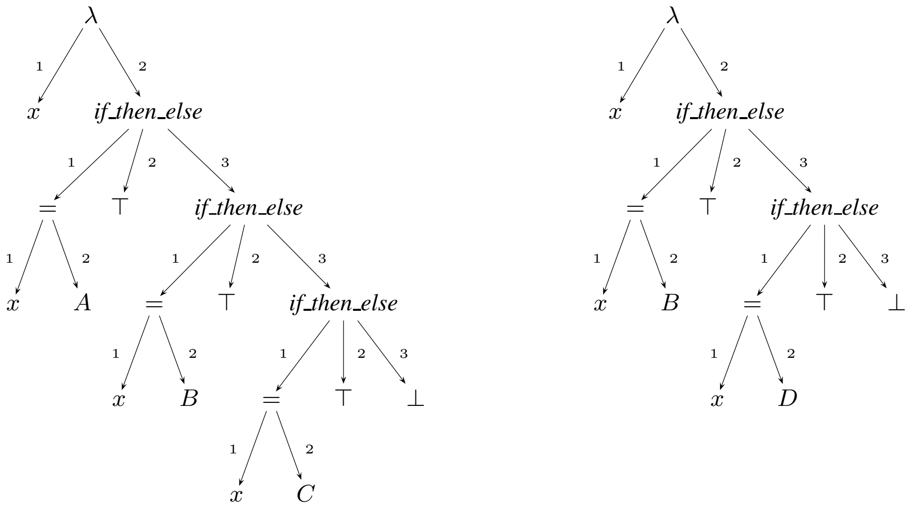


Figure 3: Term-based indexing for basic abstractions representing sets $\{A, B, C\}$ and $\{B, D\}$, where A, B, C, D : *M*, and *M* is a nullary type constructor.

3.2.2 Type-based Indexing

To solve both of the aforementioned problems of term-based indexing, we introduce an alternative approach to indexing that, instead of defining an occurrence set over all subterms of a term, defines a *type tree index set* over all subtypes of the type of a basic term. To do this we first adopt the definition of a type tree from [10] and then define a specific annotation of the tree such that every member of the type tree index set identifies a set of terms rather than a single term. The set of terms identified is guaranteed to consist entirely of basic terms. Moreover, this type-based indexing overcomes the indexing locality problem of term-based indexing.

To achieve this we follow the same interpretation of subtypes as [12] and restrict our attention to basic terms whose basic structures are in canonical form as defined below.

Definition 3.7 (Basic Structures in Canonical Form) *A type $\tau = T\ \alpha_1 \dots \alpha_k$ is a basic structure in canonical form when, for all data constructors $C_i : \tau_{i1} \to \cdots \to \tau_{in} \to \tau$ that are associated with T, all the types of τ_{ij} are subtypes of τ.*

We begin our definition of the type tree index set with some preparatory notation. Let $\mathbb{Z}^+$ denote the set of positive integers and $(\mathbb{Z}^+)^*$ the set of all strings over the alphabet of positive integers, with ε denoting the empty string. io denotes the string concatenation of i with o where $i \in \mathbb{Z}^+$ and $o \in (\mathbb{Z}^+)^*$.

Definition 3.8 (Type Tree Index Set) *The* type tree index set *of a canonical type τ, denoted $\mathcal{O}(\tau)$, is the set of strings in $(\mathbb{Z}^+)^*$ defined inductively on the structure of τ.*

1. *If τ is an atomic type, then $\mathcal{O}(\tau) = \{\varepsilon\}$.*
2. *If τ is a basic structure type $\tau = T\ \alpha_1 \dots \alpha_n$ in canonical form, with data constructors $C_i : \tau_{i_1} \to \cdots \to \tau_{i_m} \to \tau$ for all $i \in \{1, \dots, l\}$, then $\mathcal{O}(\tau) = \{\varepsilon\} \cup \bigcup_{v=1}^{p} \{ vo_v \mid o_v \in \mathcal{O}(\xi_v)\}$, where $\xi_1, \dots, \xi_p$ are the types from α_k where $\alpha_k = \tau_{i_j}$ and $\tau_{i_j} \neq \tau$, and assuming that for every $\tau_{i_j} \neq \tau$ there exists an α_k such that $\alpha_k = \tau_{i_j}$.*
3. *If τ is a basic abstraction type $\beta \to \gamma$, then $\mathcal{O}(\tau) = \{\varepsilon\} \cup \{1o \mid o \in \mathcal{O}(\beta)\} \cup \{2o \mid o \in \mathcal{O}(\gamma)\}$.*
4. *If τ is a basic tuple type $\tau = \tau_1 \times \cdots \times \tau_n$, then $\mathcal{O}(\tau) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{ io_i \mid o_i \in \mathcal{O}(\tau_i)\}$.*

Part 1, the base case, states that types for which all the associated data structures have arity zero, such as Ω (the type of the booleans), *Int* (the type of the integers), and *Char* (the type of characters), have a singleton index set containing the empty string. Part 2 states that each subtype that occurs in the signatures of the associated data constructors, and that is not itself of type τ of the basic structure, is labelled with a unique string. Part 3 labels the β and γ types of basic abstractions with a pair of unique strings. Similarly, part 4 labels each tuple item in a basic tuple with a unique string.

Definition 3.9 (Subtype at a given type tree index) *The subtype of a canonical type τ at type tree index $o \in \mathcal{O}(\tau)$, denoted $\tau]_o$, is the type that occurs in τ at o.*

The significance of defining indexing on the type tree of basic terms rather than on the terms themselves is that each member of a type tree index set $o \in \mathcal{O}(\tau)$ is not uniquely tied to any individual term of type τ. This increases the generality of the indexing such that each member of the type tree index set for type τ identifies, for any basic term $t : \tau$, an equivalence class of subterms rather than a single term. Thus $\mathcal{O}(\tau)$ induces a set of equivalence classes on the subterms of t. We refer to the set of subterms identified with a given member (index) of the type tree index set as the *basic subterm set* at that index.

Definition 3.10 (Basic Subterm Set) *If t is a basic term of type τ and $o \in \mathcal{O}(\tau)$ then the* basic subterm set of t at type tree index o, *denoted $t|_o$, is defined inductively on the length of o as follows.*

1. *If $o = \varepsilon$, then $t|_o = \{t\}$.*
2. *If $o = jo'$, for some o', and t has the form $C\ t_1 \dots t_m$, with associated type $T\ \alpha_1 \dots \alpha_n$, then $t|_o = s_j|_{o'}$ where $s_j = t_i : \tau_i$ such that $\tau_i \neq \tau$ and $\tau_i = \alpha_j$.*
3. *If $o = 1o'$, for some o', and t has the form* if_then_else(u, v, s), *then $t|_o = u|_{o'} \cup s|_o$.*
4. *If $o = 2o'$, for some o', and t has the form* if_then_else(u, v, s), *then $t|_o = v|_{o'} \cup s|_o$.*
5. *If $o = io'$, for some o', and t has the form $(t_1, \dots, t_n)$, then $t|_o = t_i|_{o'}$, for $i = 1, \dots, n$.*

A basic subterm set *is a set of basic subterms of a basic term at some type tree index. A basic subterm is* proper *if it is not at type tree index ε.*

Basic subterms indexed in part 1, the base case, are singleton sets containing an atomic term. Basic subterms indexed in part 2 are basic structures. Basic subterms indexed in parts 3 and 4 are the support and value of basic abstractions, i.e. respective instances of α and β, from $\alpha \to \beta$. Basic subterms indexed in part 5 are basic tuples.

Below we give examples of a type tree index set and basic subterm sets for each of basic tuples, basic structures, and basic abstractions. Starting with basic tuples in Example 3.11 where it can be seen that in comparison to the term-based indexing from Example 3.4, type-based indexing identifies all the same terms, but as singleton sets and in addition it identifies the reflexive term at $t|_\varepsilon$.

Example 3.11 *If basic tuple $t \in \mathfrak{B}_{M\times N\times O\times P}$ is the term $t = (A, B, C, D)$, where $A : M,\ B : N,\ C : O,\ D : P$, then the type tree index set of t is $\mathcal{O}(t) = \{\varepsilon, 1, 2, 3, 4\}$, the derivation of which can be seen from Figure4. The basic subterm sets of t are $t|_\varepsilon = \{(A, B, C, D)\}$, $t|_1 = \{A\}$, $t|_2 = \{B\}$, $t|_3 = \{C\}$ and $t|_4 = \{D\}$.*

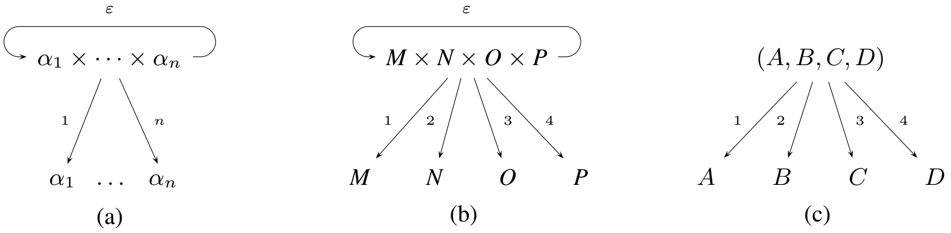


Figure 4: Type-based indexing for basic tuples. (a) Type tree index for n-tuples of type $\alpha_1 \times \cdots \times \alpha_n$. (b) Type tree index for 4-tuples of type $M \times N \times O \times P$. (c) Basic subterm tree for term (A, B, C, D) where $A : M,\ B : N,\ C : O,\ D : P$.

Representing basic structures, the usual right branching representation of lists is given in Example 3.12, where the basic subterm set at $t|_1$ captures one meaning of a list as a set of values and $t|_\varepsilon$ captures the meaning of a list as a set of sequences.

Example 3.12 *If τ is a type of lists such that $\tau = List\,M$, where $M \subseteq \mathfrak{B}$ is a nullary type constructor, with associated data constructors $\#$ and $[]$, having signatures $[] : List\,M$, and $\# : M \to List\,M \to List\,M$, then the type tree index set of τ is $\mathcal{O}(\tau) = \{\varepsilon, 1\}$. If basic terms $s, t \in \mathfrak{B}_{List\,M}$ are the lists $s = [A, B, C]$ and $t = [A, D]$, then as can be seen from Figure6, the basic subterm sets of s and t are $s|_\varepsilon = \{[A, B, C], [B, C], [C], []\}$, $s|_1 = \{A, B, C\}$, and $t|_\varepsilon = \{[A, D], [D], []\}$, $t|_1 = \{A, D\}$.*

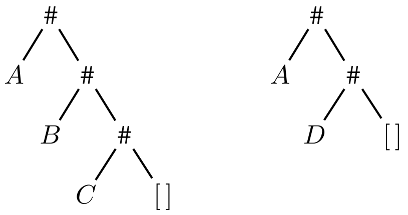


Figure 5: Lists $[A, B, C]$ and $[A, D]$ as basic structures.

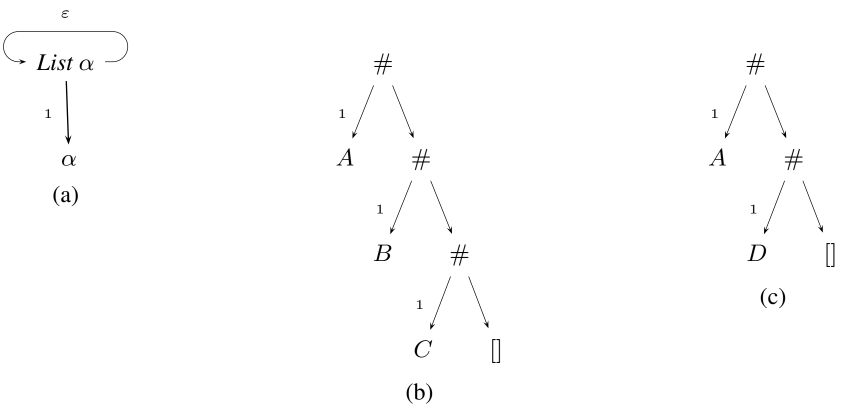


Figure 6: Type-based indexing for basic structures. (a) Type tree index for *List* α. (b) and (c) Basic subterm trees for terms $[A, B, C]$ and $[A, D]$ of type *List M* where A, B, C, D : *M*.

For basic abstractions, a set is given in Example 3.13 and a multiset in Example 3.14. For both sets and multisets, $t|_1$ captures the meaning as a set of values whereas $t|_2$ will always be $\{\top\}$ for sets and a set of multiplicities for multisets. A corollary of Definition 3.8 is that the type tree index set of a basic abstraction type is always $\{\varepsilon, 1, 2\}$.

Example 3.13 *If τ is a basic abstraction type representing sets such that $\tau = M \rightarrow \Omega$, where $M \subseteq \mathfrak{B}$ is a nullary type constructor, then the type tree index set of τ is $\mathcal{O}(\tau) = \{\varepsilon, 1, 2\}$. If basic term $t = \{A, B, C\}$, where A, B, C : M, then the basic subterm sets are $t|_\varepsilon = \{\{A, B, C\}\}$, $t|_1 = \{A, B, C\}$ and $t|_2 = \{\top\}$.*

Example 3.14 *If τ is a basic abstraction type representing multisets such that $\tau = M \rightarrow$ Nat, where $M \subseteq \mathfrak{B}$ is a nullary type constructor and Nat is the type of the natural numbers, then the type tree index set of τ is $\mathcal{O}(\tau) = \{\varepsilon, 1, 2\}$. If basic term $t = \{A, A, A, B, C, C\}$, where A, B, C : M, then the basic subterm sets are $t|_\varepsilon = \{\{A, A, A, B, C, C\}\}$, $t|_1 = \{A, B, C\}$ and $t|_2 = \{1, 2, 3\}$.*

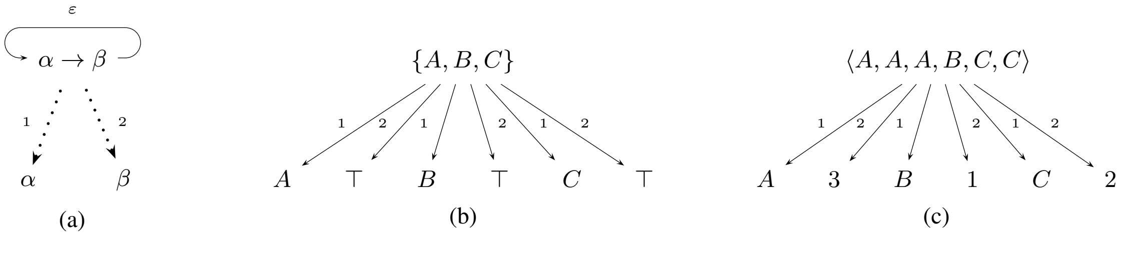


Figure 7: Type-based indexing for basic abstractions. (a) Type tree index for type $\alpha \rightarrow \beta$. (b) Basic subterm tree for set $\{A, B, C\}$, type $M \rightarrow \Omega$, where A, B, C : *M* and $\top$: Ω. (c) Basic subterm tree for multiset $\langle A, A, A, B, C, C \rangle$, type $M \rightarrow$ Nat, where A, B, C : *M* and $1, 2, 3$: Nat.

Proposition 3.15 *If τ is a basic abstraction type such that $\tau = \alpha \rightarrow \beta$, where $\alpha, \beta \in \mathfrak{B}$, then the type tree index set of τ is $\mathcal{O}(\tau) = \{\varepsilon, 1, 2\}$.*

Proof 3.16 *The result is a corollary of part 3 of Definition 3.8.*

3.2.3 Type Name-based Indexing

A useful and straight forward reformulation of type-based indexing is *type name-based indexing* that, instead of enumerating the edges of the type tree, directly labels the vertices of the type tree. The simplest approach being to assign a unique type name to every vertex in the type tree. If the names assigned have no understandable meaning to humans then this method offers no advantages over type-based indexing. However, if the knowledge representational formalism used to define types and data instances uses human-understandable names then type name-based indexing provides a useful notation for referring to basic subterm sets, as illustrated in Example 3.17.

Example 3.17 *Let Author be the type of authors from the publications domain, which define declaratively in the Haskell style syntax from [8] as follows.*

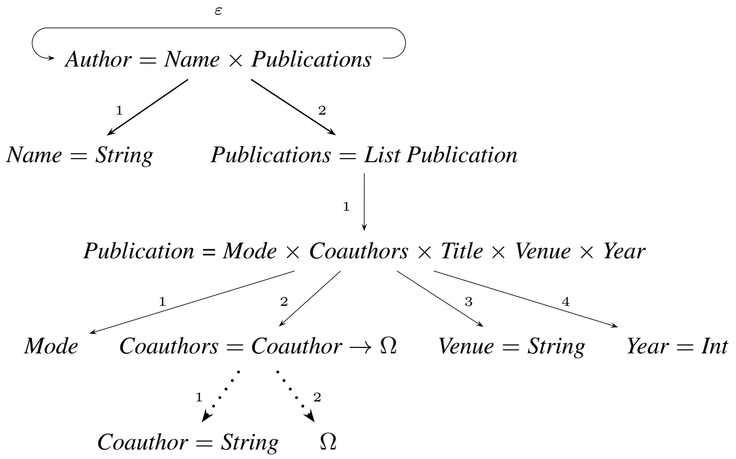
```
type Author = (Name,Publications);
type Name = String;
type Publications = List Publication;
type Publication = (Mode,Coauthors,Title,Venue,Year);
data Mode = Journal | Proceedings | ... | Book;
type Coauthors = Coauthor -> Bool;
type Coauthor = String;
type Title = String;
type Venue = String;
type Year = Int;
```

This states that Author is a pair of Name and Publications, where Name is an alias for String the type of strings, and Publications is a list of publications, which in turn is a 5*-tuple of Mode, Coauthors, . . . , Year, where Mode has the nullary data constructors Journal, Proceedings, . . . , Book, and so on through to Year which is an alias for the type Int, the type of the integers. Coauthors is a basic abstraction from Coauthor to Bool, where Bool is the type Ω, i.e. Coauthors is a set of coauthors. To ensure the required uniqueness of type names Coauthor, Title, Venue and Name are aliases for the type String. The type tree index set is thus*

$$\{\, Author,\ Author.Name,\ \ldots,\ Author.Publications.Publication.Year \,\}.$$

A type tree index set generated using this method is isomorphic with that produced by Definition 3.8, as illustrated informally in Figure8. The constraint that all basic subtypes must be uniquely named permits the following simpler definition of a basic subterm set.

Definition 3.18 (Basic Subterm Set (with named types)) *If t is a closed basic term of type τ and $\alpha \subseteq \tau$ then the* basic subterm set of t at type α, *denoted $t|_\alpha$, is $t|_\alpha = \{s \mid s$ occurs in t with type $\alpha\}$. A* basic subterm set *is a set of basic subterms of a basic term at some type tree $\alpha \subseteq \mathfrak{B}$. A basic subterm is* proper *if $\alpha \neq \tau$.*



ε
Author = Name × Publications
1
2
Name = String
Publications = List Publication
1
Publication = Mode × Coauthors × Title × Venue × Year
1
2
3
4
Mode
Coauthors = Coauthor → Ω
Venue = String
Year = Int
1
2
Coauthor = String
Ω

Figure 8: Type name-based and type-based indexing for type *Author*.

4 Basic Term Relational Model

We now upgrade the relational model for structured data. The way we achieve this is to first upgrade the knowledge representation of the relation to be a set of basic terms rather than the traditional set of tuples. We then upgrade the relation index so that it indexes parts of a basic term rather than the traditional parts of a tuple. Once these two steps are completed, upgrading the relational operations follows almost automatically with only modest changes to the definitions of the θ-restriction and joins. So to begin, we first upgrade the relation from section 2.1 to become the *basic term relation*.

4.1 Basic Term Relational Representation

Definition 4.1 (Basic Term Relation) *A* basic term relation $R \subseteq \mathfrak{B}_\alpha$ *is a finite set of basic terms for some given type* $\alpha \in \mathfrak{S}^c$*.*

In the above definition of the basic term relation we describe R as a subset of $\mathfrak{B}_\alpha$ in order to emphasise the intended meaning of R as a set. Doing so also maintains a clear syntactic similarity to the definitions from the traditional relational algebra. However, in the higher-order logic we could equally have written $R \in \mathfrak{B}_{\alpha\to\Omega}$ because, being a set, R is a basic abstraction of type $\alpha \to \Omega$, where Ω is the type of the booleans. Thus a basic term relation is itself a basic term.

Having upgraded our representation of a relation $R : \tau$ to handle structured data represented as basic terms, and having chosen a suitable indexing method for the basic subterm set $\mathcal{O}(\tau)$, we are now able to conveniently define the *basic term relation index* as the structured data counterpart of the relation index.

Definition 4.2 (Basic Term Relation Index) *The* basic term relation index I_R *of a basic term relation* R *of type* τ *is* $I_R = \mathcal{O}(\tau)$*.*

4.2 Basic Term Relational Operations

4.2.1 Fundamental Basic Term Relational Operations

The basic term relational algebra defines five operations that directly correspond to the same five fundamental operations defined in relational algebra.

- basic term union ($\cup$)
- basic term difference ($-$)
- basic term product ($\times$)
- basic term projection (π)
- basic term restriction (σ)

Each of these operations on basic term relations is defined and discussed throughout the remainder of this section. The definitions of basic term union and difference (and, later when we discuss the non-fundamental operations, intersection) accord the usual set theory but apply solely to sets that are basic term relations and are relations of the same type.

Definition 4.3 (Basic Term Union) *The* basic term union $A \cup B$ *of basic term relations* $A \subseteq \mathfrak{B}_\alpha$ *and* $B \subseteq \mathfrak{B}_\alpha$*, for some* $\alpha \in \mathfrak{S}^c$*, is the basic term relation* $A \cup B = \{t \mid t \in A \vee t \in B\}$*.*

Definition 4.4 (Basic Term Difference) *The* basic term difference $A - B$ *of basic term relations* $A \subseteq \mathfrak{B}_\alpha$ *and* $B \subseteq \mathfrak{B}_\alpha$*, for some* $\alpha \in \mathfrak{S}^c$*, is the basic term relation* $A - B = \{t \mid t \in A \wedge t \notin B\}$*.*

The basic term product is, potentially, more troublesome to define than its corresponding operation from traditional relational algebra in that there is no single natural structure for combining an arbitrary pair of tuples that is the analogue of concatenating a pair of relational tuples into a single relational tuple. As no one structure seems more appropriate than another here, we simply adopt the traditional mathematical Cartesian product as this is both sufficient and straight forward.

Definition 4.5 (Basic Term Product) *The basic term product $A \times B$ of basic term relations $A \subseteq \mathfrak{B}$ and $B \subseteq \mathfrak{B}$ is their Cartesian product such that $A \times B = \{ (\ a, b\ ) \mid a \in A,\ b \in B\}$.*

Note that duplicates are removed from the basic term relational product, as is normal for sets, and so $|A \times B| \leq |A||B|$. Also, there is no requirement for A and B to be of the same type; if the type of A is α and of B is β then the type of $A \times B$ is $\alpha \times \beta$.

Definition 4.6 (Basic Term Projection) *If $t \in \mathfrak{B}$ then the* basic term projection *π of t on $i \in I_t$ is*

$$\pi_i(t) = \{s \mid s \textit{ is the basic subterm of } t \textit{ at type tree index } i\}.$$

A basic term projection $\pi_i(t)$ may also be written as $t|_i$.

Basic term projection is defined over basic terms rather than just a basic term relation and so is more general than relational projection from traditional relational algebra. Thus, the same basic term projection operation may be applied to both an entire basic term relation or to an individual member of a basic term relation.

Definition 4.7 (Basic Term θ-Restriction) *Let θ be a predicate $\theta : (\mathfrak{B}_\alpha \to \Omega) \to (\mathfrak{B}_\alpha \to \Omega) \to \Omega$ for some $\alpha \in \mathfrak{S}^c$. If A and B are basic term relations with basic terms $a|_i \subseteq \mathfrak{B}_\alpha$ and $b|_j \subseteq \mathfrak{B}_\alpha$ respectively for some $(i, j) \in I_A \times I_B$, then* basic term θ-restriction *$\sigma_{i\theta j}$ is defined on $T \subseteq A \times B$ as*

$$\sigma_{i\theta j}(T) = \{(a, b) \mid\ a|_i\, \theta\, b|_j\ \wedge\ (a, b) \in T\}.$$

The predicate $\theta : (\mathfrak{B}_\alpha \to \Omega) \to (\mathfrak{B}_\alpha \to \Omega) \to \Omega$ is defined on sets of basic terms. In other words, θ is a binary predicate on basic term relations. The basic term restriction, like its counterpart in traditional relational algebra, produces relations of the same type as the type to which it is applied and of cardinality such that $|\sigma_{i\theta j}(T)| \leq |T|$.

Definition 4.8 (Basic Term Generalised Restriction) *Let φ be a proposition that consists of atoms as allowed in basic term θ-restriction and the logical operations $\wedge$, $\vee$ and $\neg$. If A and B are basic term relations then the* basic term generalised restriction *σ_φ is defined on $T \subseteq A \times B$ as*

$$\sigma_\varphi(T) = \{\ t \mid \varphi(t)\ \wedge\ t \in T\}.$$

4.2.2 Derivative Basic Term Relational Operations

The remaining three operations on basic term relations are also counterparts of their corresponding operations in traditional relational algebra and may be defined solely in terms of the five fundamental basic term relational operations.

- basic term intersection ($\cap$)
- basic term division ($\div$)
- basic term join ($\bowtie$)

In the remainder of this section we define and discuss each of these non-fundamental operations and explain why they are conceptually useful in their own right.

Definition 4.9 (Basic Term Intersection) *The* basic term intersection *$A \cap B$ of basic term relations $A \subseteq \mathfrak{B}_\alpha$ and $B \subseteq \mathfrak{B}_\alpha$, for some $\alpha \in \mathfrak{S}^c$, is the basic term relation $A \cap B = \{t \mid t \in A \wedge t \in B\}$.*

Expressed in terms of the fundamental relational operations $A \cap B = A - (A - B)$, $A \cap B = B - (B - A)$ and $A \cap B = A \cup B - (A - B) - (B - A)$.

Basic term division is the reverse of basic term product[3].

Definition 4.10 (Basic Term Division) *The* basic term division *$A \div B$ of basic term relations $A \subseteq \mathfrak{B}_\alpha \times \mathfrak{B}_\beta$ and $B \subseteq \mathfrak{B}_\beta$, for some $\alpha \in \mathfrak{S}^c$, is the basic term relation*

$$A \div B = \{a \mid \forall b \in B, (a, b) \in A\}.$$

Expressed in terms of the fundamental relational operations,

$$A \div B = \pi_1(A) - \pi_1((\pi_1(A) \times B) - A).$$

[3] We denote division as '$\div$' rather than '/' to avoid confusion with the latter's use in *parameter/type* binding notation from type substitutions in the higher-order logic.

Recall that the type tree index set for a pair (a, b) is $\{\varepsilon, 1, 2\}$ with the corresponding basic subterm set $\{(a, b), a, b\}$. Hence the projection π of (a, b) on 1 is $\pi_1 = (a, b)|_1 = a$.

Definition 4.11 (Basic Term θ-Join) *Let $\theta : (\mathfrak{B}_\alpha \to \Omega) \to (\mathfrak{B}_\alpha \to \Omega) \to \Omega$ be a predicate for some type $\alpha \in \mathfrak{S}^c$. If A and B are basic term relations with basic terms $a|_i \subseteq \mathfrak{B}_\alpha$ and $b|_j \subseteq \mathfrak{B}_\alpha$ respectively for some $(i, j) \in I_A \times I_B$ then the* basic term θ-join $\bowtie_{i\,\theta j}$ *of A and B is defined as*

$$A \bowtie_{i\,\theta j} B = \sigma_{i\,\theta j}(A \times B).$$

Note that the basic term θ-join, unlike its traditional relational counterpart, does not include a projection on the selection.

Definition 4.12 (Basic Term Generalised Join) *Let φ be a proposition that consists of atoms as allowed in basic term θ-restriction and the logical operations $\wedge$, $\vee$ and $\neg$. If A and B are basic term relations then the* basic term join $\bowtie_\varphi$ *is defined as*

$$A \bowtie_\varphi B = \sigma_\varphi(A \times B).$$

The closeness in form of the definition of the basic term join to that of the relational join facilitates the following result.

Proposition 4.13 *Relational joins are a special case of basic term relational joins.*

Proof 4.14 *Assume relation $R \subseteq D_1 \times \cdots \times D_n$ for some domains $D_1, \ldots, D_n$. Assume appropriate type constructors and data constructors such that $D_1, \ldots, D_n \subseteq \mathfrak{B}$. Let basic term relation $S \subseteq D_1 \times \cdots \times D_n$. Let I_R be the relation index of R and I_S be the basic term relation index of S. Clearly there is a surjection from I_R into I_S and thus from the set of tuple items in each tuple in R to the set of subterms in each corresponding basic term tuple in S. Assume the θ operators are available for basic terms and the result follows.*

4.3 Basic Term Relational Algebra

Given the basic term representation and fundamental basic term operations defined above we can now define the *basic term relational algebra* closely mirroring our earlier definition the relational algebra.

Definition 4.15 (Basic Term Relational Algebra) *Let $\mathcal{D}$ be a collection of non-empty domains $\{\mathfrak{B}_\alpha\}_{\alpha \in \mathfrak{S}^c}$. Let $\boldsymbol{R}$ be a set of basic term relations such that $\boldsymbol{R} = \{R \mid R \subseteq \mathfrak{B}_\alpha, \alpha \in \mathfrak{S}^c \ \wedge \ \mathfrak{B}_\alpha \in \mathcal{D}\}$. Let ind be a total function from basic term relations $R \in \boldsymbol{R}$ to basic term relation indexes $I_R \subseteq \mathbb{N}^+$. Let Θ be a set of predicates over domains in $\mathcal{D}$, and the logical operations $\wedge$, $\vee$ and $\neg$ over the booleans. Let $\boldsymbol{O}$ be the set of operations: basic term union $\cup$, basic term difference $-$, basic term product $\times$, basic term projection π, and basic term restriction σ. The* basic term relational algebra *over $\mathcal{D}$, dom, $\boldsymbol{R}$, ind, Θ and $\boldsymbol{O}$ is the 6-tuple $\mathscr{R} = (\mathcal{D}, dom, \boldsymbol{R}, ind, \Theta, \boldsymbol{O})$. An* algebraic expression *over $\mathscr{R}$ is any expression formed legally, according to the definitions of the operations, from the relations in $\boldsymbol{R}$ using the operations in $\boldsymbol{O}$.*

5 Summary and Future Work

The relational model that underpins SQL databases systems is not well suited to the representation of many commonly occurring unstructured, semi-structured or structured data types. The individuals as terms representation, using basic terms from the higher-order logic, facilitates representation of such heterogeneous data types without the convoluted schemes entailed in their relational representation. The relational algebra for basic terms introduced in this paper has the potential to offer similar convenience to the original relational algebra, but exchanging data normalisation for increased representational power. The subterm indexing methods presented here could be further developed to, for example, admit XPath-like traversal of arbitrary data. Also, given the syntactic and semantic heterogeneity of data from diverse sources, there is considerable scope for approximate, rather than exact, basic term joins. Potential application areas are in the de-normalised settings of NoSQL and Big Variety Big Data.

References

[1] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[2] E. F. Codd. Derivability, redundancy, and consistency of relations stored in large data banks. Technical Report JR599, IBM, San Jose, 1969.

[3] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[4] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems, Vol. 4, No. 4, December 1979.*, 4(4):397–434, December 1979.

[5] E. F. Codd. *The Relational Model for Database Management, Version 2*. Addison Wesley, 1990.

[6] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.

[7] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 5th edition, March 2006.

[8] Thomas Gaertner, John W. Lloyd, and Peter A. Flach. Kernels and distances for structured data. *Machine Learning*, 57(3):205–232, December 2004.

[9] Herve Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 16(2):153–185, 1984.

[10] Elias Gyftodimos and Peter A. Flach. Combining bayesian networks with higher-order data representations. In *Proceedings of the 6th International Symposium on Intelligent Data Analysis (IDA'06)*, pages 145–157. Springer-Verlag, September 2005.

[11] John W. Lloyd. Higher-order computational logic. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2407 of *Lecture Notes in Computer Science*, pages 105–137. Springer, 2002.

[12] John W. Lloyd. *Logic and Learning*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[13] David Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, Md., USA, 1983.

[14] Simon Price and Peter Flach. Querying and merging heterogeneous data by approximate joins on higher-order terms. In *Inductive Logic Programming, 18th International Conference, ILP 2008, Prague, Czech Republic, September 2008, Proceedings*, volume 5194 of *LNCS/LNAI*, pages pp. 226–243. Springer-Verlag Berlin Heidelberg, 2008.