Deductive query languages for recursively typed complex objects

Richard Hull \textsuperscript{a,2}, Jianwen Su \textsuperscript{b,\*3}

\textsuperscript{a} Bell Laboratories, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974, USA
\textsuperscript{b} Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA 93106, USA

Received 1 March 1994; received in revised form 4 February 1997; accepted 12 September 1997

Abstract

Deductive database query languages for recursively typed complex objects based on the set and tuple constructs are studied. A fundamental characteristic of such complex objects is that in them, sets may contain members with arbitrarily deep nesting of tuple and set constructs. Relative to mappings from flat relations to flat relations, two extensions of COL in this context (with stratified semantics and inflationary semantics, respectively) are shown to have the expressive power of computable queries. Although the deductive calculus of Bancilhon and Khoshafian has the ability to simulate Turing machines, when restricted to flat input and output its expressive power is characterized by a weak variant of the conjunctive queries. © 1998 Elsevier Science Inc. All rights reserved.

1. Introduction

In database management systems, data models and their associated query languages provide a means to organize and manipulate large quantities of data. As a
pioneer model, the relational data model [20] suits business applications where data can be represented mostly as tables or relations. However, other applications such as software engineering, CAD/CAM, and scientific applications demand richer models and languages since their data are too complex to fit into relations. Among the important extensions to the relational data model is the set (or grouping) construct that arises: (i) in the non-first normal form relations or complex objects with calculus-like query languages [37,47,3], with relational algebra extensions [24,1,50,21,3], or with deductive languages [41,40,14,5], and (ii) in semantic data models [31,49,6,42] (see also [30]). In summary, there are two fundamentally different ways to incorporate sets into database structures:

1. Non-recursive types, in which case all elements in the domain of a given type have bounded nesting depth of the tuple and set constructs. Examples include the models that support homogeneous sets [24,1,50,47,41,3,6,42] (see also [35]), or finitely heterogeneous sets, as in COL [5].

2. Recursive types, which permit elements in the domain of a given type to have arbitrarily deep nesting depth of the constructs. This approach was pioneered in Database Logic [37], and used in a number of investigations, including e.g., LDL [14], FAD 4 [10], the deductive calculus of Bancilhon and Khoshafian [13], the "Set Theoretic Data Model" of Gemstone [19,45], and also the "directory query language" DL of [21].

In the database community, query languages for complex objects with non-recursive types were extensively studied [3,33,32,43,46,26-29] and languages with recursive types were discussed in [21,34] but not fully investigated. In the companion paper [34], we presented the results concerning algebraic and calculus languages with recursive types. In this paper, we study deductive query languages with recursive types.

Relational database queries were originally introduced by Chandra and Harel [16,17] as mappings from relational databases to relations that are "generic", i.e., commute with the permutations of the underlying domain. The notion was naturally generalized to complex objects in [33] where it is shown that complex object languages express exactly the set of "elementary" queries, i.e., queries of hyper-exponential data complexity [51]. The query notion was further extended to databases with recursive types independently in [34,21]. Dahlhaus and Makowsky [21] considered directories, which are databases over a single universal recursive type, and queries over them and proved that an extension DL of Chandra and Harel's QL [16] to directories expresses exactly all computable directory queries. On the other hand, [34] used an extension of the complex object model [3] with a universal recursive type and showed that: (1) recursive types do not increase expressive power of the complex object algebra but raises the expressive power from elementary queries computable queries once the loop construct 5 is added; and (2) the complex object calculus with recursive types has expressive power equivalent to the arithmetical hierarchy [48]. By focusing on the deductive languages for complex objects, this paper provides an interesting complement to the results in [34]. The results reported both here and in [34] confirm that adding recursive types yields considerably more expressive power.

---

4 Recursive types were later eliminated in FAD [22,23].
5 The loop construct can be simulated by the powerset operator in the context of complex objects [29].
In the formal development of the paper, we focus on a very general family of recursively defined types, called \textit{r-types}, which generalize that of complex object types of [3,33]. In our setting, essentially any recursive complex object type is subsumed by some \textit{r-type}; our characterizations thus provide upper bounds on the expressive power of languages supporting essentially arbitrary frameworks for recursively typed complex objects. On the other hand, all of the characterizations of expressive power are robust in the sense that they continue to hold for all recursive complex object typing disciplines discussed in the database literature.

Query languages provide notations for expressing database queries. It is important to provide the user with declarative languages. The failure of having an effective semantics of the calculus for recursively typed complex objects naturally leads to the question on deductive languages in this context. In the literature, several deductive languages incorporating the set construct have been suggested: COL [5], BK-calculus [13], LDL [14], the recursive language of [3], etc. The first focus of this paper is to study the expressive power of two extensions of COL with recursive types under the \textit{stratified} and \textit{inflationary} semantics (respectively). We prove that both extensions express exactly the class of computable queries. For completeness, it is also stated that in the case of non-recursive types, COL under inflationary semantics has the same expressive power as under stratified semantics. Analogous results hold for the recursive language of [3]. This provides an interesting contrast to the fact that in the relational model the stratified DATALOG with negation is strictly weaker than inflationary DATALOG with negation [38,39,9]. We also note here that for DATALOG with counting the two semantics again yield the same expressive power [25]. Recursive types increase the expressive power of query languages in a way similar to "invented values" [33]. More recently, invented values in the deductive language ILOG [36] were studied in [15].

The proofs of the characterization results for COL extensions to recursive types use a technical tool called "domain Turing machine" developed in [34]. Briefly, a domain Turing machine differs from a conventional Turing machine by allowing the input alphabet to be infinite and having a register. Using two generic symbols, the transition function can still be finitely described. By the equivalence between domain Turing machines and conventional Turing machines established in [34], the use of the former with infinite alphabets greatly simplifies the proofs.

The second focus of this paper is on the deductive language BK-calculus of [13] for recursively typed complex objects. The language is distinguished by its use of the \textit{sub-object} relationship instead of \textit{equality (=)} in defining the semantics of rule application. As a result, the techniques appropriate for studying its expressive power are significantly different from that for the other languages with equality-based semantics. On one hand, when provided with suitably encoded input, we prove that the BK-calculus can simulate arbitrary Turing machines. Thus the data complexity is unbounded. On the other hand, when restricted to relational input and output, we show that the expressive power of the BK-calculus is equivalent to a natural variant of the conjunctive queries of [18]. More interestingly, this variant is strictly weaker than the conjunctive queries; in particular, it cannot compute the natural join.

The paper is organized as follows. Section 2 introduces basic concepts of the model of recursively typed complex objects, the notion of a query and the two classes of queries: computable and elementary queries, and domain Turing machines. Section 3 presents definitions and results for extension of COL with recursive types as
well as COL for non-recursive types under the inflationary semantics. The BK-calculus is presented and studied in Section 4. Conclusions are included in Section 5.

2. Preliminaries

In this section, we define the "complex object model" with recursive types, "query functions", and the classes of "computable and elementary queries". We also briefly review "domain Turing machines" originally introduced in [34], which are used in proving the results on deductive languages. The presentations of terminology for COL and the BK-calculus are in Sections 3 and 4 (respectively).

We assume that \( U \) and \( P \) are two disjoint countably infinite sets of atomic objects and predicate names (respectively). To introduce recursively typed complex objects, we define the following single "unrestricted" recursive type, which can be used as "leaves" in types which are otherwise non-recursive. The results obtained here generalize to essentially any family of types that has at least one recursive type.

Let \( O \) denote the smallest set such that:

1. \( U \subseteq O \);
2. \( \{o_1, \ldots, o_n\} \in O \) if \( n \geq 0 \) and \( o_i \in O \) for each \( 1 \leq i \leq n \); and
3. \( [o_1, \ldots, o_n] \in O \) if \( n \geq 1 \) and \( o_i \in O \) for each \( 1 \leq i \leq n \).

The equality between objects in \( O \) is defined naturally. The type system is defined as follows.

**Definition 2.1.** The family of r-types is defined recursively as follows:

1. \( U \) is the atomic r-type;
2. \( O \) is the universal r-type;
3. \( \{\} \) is a set r-type if \( \tau \) is an r-type; and
4. \( [\tau_1, \ldots, \tau_k] \) is a \( k \)-ary tuple r-type if \( k \geq 1 \) and \( \tau_i \) is an r-type for each \( 1 \leq i \leq k \).

Two r-types are equal if they are the same syntactic object. An r-type is flat if it is a \( k \)-ary tuple r-type of the form \( [U, \ldots, U] \), which is also denoted as \( U^k \). An r-type is non-recursive if it does not contain the r-type \( O \).

By the definition, non-recursive r-types correspond to complex object types of \([3,33,28]\).

**Definition 2.2.** The domain of an r-type \( \tau \), denoted \( \text{dom}(\tau) \), is defined recursively by:

1. \( \text{dom}(U) = U \);
2. \( \text{dom}(O) = O \);
3. \( \text{dom}(\{\}) = \{\{o_1, \ldots, o_n\} \mid n \geq 0 \) and \( o_i \in \text{dom}(\tau) \) for each \( 1 \leq i \leq n \}; \text{ and}
4. \( \text{dom}([\tau_1, \ldots, \tau_n]) = \{[o_1, \ldots, o_n] \mid o_i \in \text{dom}(\tau_i) \) for each \( 1 \leq i \leq n \}. \)

Each element in \( \text{dom}(\tau) \) is an object of r-type \( \tau \). Any finite subset of \( \text{dom}(\tau) \) is an instance of \( \tau \). Let \( \text{inst}(\tau) \) denote the family of all instances of \( \tau \).

**Example 2.3.** Let \( a, b, c \), be constants in \( U \). The following are objects of r-type \( \tau = [U, O] \):

\[[a, a], \quad [b, [a, a]], \quad [c, [b, [a, a]]]]\]
but these are not: \(a, [a], [a, b, c].\) Let \(\alpha = a_1a_2 \cdots a_n\) be a finite string over \(U\) where \(n > 0.\) The following presents three naive ways to encode \(\alpha\) into an object of r-type \(\tau, O\) and \(\{O\}\) (respectively):

\[
[a_1, [a_2, \ldots, [a_n]]], \quad [a_1, a_2, \ldots, a_n], \quad \{a_1, \{a_2, \ldots, \{a_n\}\}\}.
\]

With the above definitions, we now define database schemas and instances.

**Definition 2.4.** A database schema is a sequence \(D = \langle p_1: \tau_1, \ldots, p_n: \tau_n \rangle\) such that \(p_1, \ldots, p_n\) are distinct predicate names in \(P,\) and \(\tau_1, \ldots, \tau_n\) are r-types. The schema \(D\) is flat if each \(\tau_i (1 \leq i \leq n)\) is flat. A (database) instance of \(D\) is a sequence \(d = \langle p_1: I_1, \ldots, p_n: I_n \rangle\) where \(I_i \in \text{inst}(\tau_i)\) for each \(1 \leq i \leq n.\) The family of all instances of \(D\) is denoted \(\text{inst}(D).\)

Finally, we define the notions of “active domain” of an instance and “constructed domain” of an r-type with respect to an instance. The former is the set of atomic objects used in building an object (or instance) of a type (or database schema), and the latter is the set of all objects built using a given set of atomic objects.

**Definition 2.5.** The active domain of an object \(o,\) denoted \(\text{adom}(o),\) is defined recursively by:

1. If \(o \in U\) then \(\text{adom}(o) = \{o\};\)
2. if \(o = [o_1, \ldots, o_n]\) is a tuple, then \(\text{adom}(o) = \bigcup_{1 \leq i \leq n} \text{adom}(o_i)\); and
3. if \(o\) is a set, then \(\text{adom}(o) = \bigcup_{o' \in o} \text{adom}(o').\)

The active domain of an instance \(I\) of an r-type \(\tau\) is defined as \(\text{adom}(I) = \bigcup_{o \in \text{adom}(\tau)} \text{adom}(o).\) The active domain of a database instance \(d = \langle p_1: I_1, \ldots, p_n: I_n \rangle\) is \(\text{adom}(d) = \bigcup_{1 \leq j \leq n} \text{adom}(I_j).\) For each finite \(X \subseteq U\) and r-type \(\tau,\) the constructed domain of \(\tau\) using \(X\) is \(\text{cons}_{\tau}(X) = \{o \in \text{dom}(\tau) \mid \text{adom}(o) \subseteq X\}.\)

In this paper, database queries are viewed as mappings from database instances to instances of types. Let \(D\) be a database schema and \(\tau\) an r-type. A database mapping \(f\) from \(D\) to \(\tau,\) denoted \(f: D \rightarrow \tau,\) is a partial mapping from \(\text{inst}(D)\) to \(\text{inst}(\tau).\) Since database queries treat data objects in an uninterpreted way, database mappings are required to be “generic” in the following sense.

**Definition 2.6.** If \(C \subseteq U\) and \(f\) is a database mapping, then \(f\) is \(C\)-generic if \(f \circ \rho = \rho \circ f\) for each permutation \(^6\rho\) of \(U\) with \(\forall x \in C, \rho(x) = x;\) \(f\) is generic if it is \(C\)-generic for some finite \(C.\)

For our purpose of investigation, we focus on those functions whose domains are flat databases (instances of flat database schemas) and ranges are flat relations (instances of flat r-types).

**Definition 2.7.** A database mapping \(f: D \rightarrow \tau\) is a query function if \(f\) is generic, \(D\) is a flat database schema, and \(\tau\) is a flat r-type.

---

\(^6\rho\) is extended naturally to databases.
We now introduce two interesting classes of queries: "computable" queries and "elementary" queries. Let the class of hyper-exponential functions \( \text{hyp}_i(n) \) be defined such that \( \text{hyp}_0(n) = n \) and \( \text{hyp}_{i+1}(n) = 2^{\text{hyp}_i(n)} \), for \( i \geq 0 \).

The family of "computable queries" was introduced in [16] and also studied in [8]. It is essentially the class of generic database mappings which are computable by Turing machines. Variations depending on classes of input and output schemas have been studied, and [9] considered both non-deterministic and deterministic mappings. The family of "elementary queries" arises naturally in the query languages for complex objects [33,43]. Roughly speaking, this family contains queries whose time (or equivalently, space) data complexity [51] are elementary functions.

**Definition 2.8.** The class \( \mathcal{C} \) of computable query functions is the set of query functions which are Turing computable. The class \( \mathcal{E} \) of elementary query functions is the set of query functions in \( \mathcal{C} \) which have hyper-exponential time (or space) data complexity.

A query (in some language) is a syntactic expression. With associated semantics, each query expression realizes a generic database mapping. As our focus is on query functions which are from flat schemas to flat r-types, we consider only query expressions with that property in the remainder of the paper.

**Definition 2.9.** Two query languages \( L_1 \) and \( L_2 \) are equivalent if they realize the same set of query functions. \( L_1 \) is no more expressive than \( L_2 \) (\( L_1 \sqsubseteq L_2 \)) if each query function realizable in \( L_1 \) is also realizable in \( L_2 \). \( L_1 \) is \( \mathcal{C}(\mathcal{E}) \)-equivalent if it realizes exactly the set of all query functions in \( \mathcal{C} \) (respectively \( \mathcal{E} \)).

The notion of \( \mathcal{C} \)-equivalence has been called "computationally complete" in other investigations. When the context is clear, both query functions and query expressions are referred simply as queries.

We shall discuss deductive languages for recursive types, like extended versions of COL [5] and BK-calculus [13]. Two extensions of COL will be introduced briefly in Section 3 which have stratified and inflationary semantics (respectively). The complex objects of [13] are not typed. The set of objects together with the "sub-object" relationship forms a lattice. BK is a rule-based language with a fixpoint semantics. Informally, a query consists of a set of rules. An application of rules is to find all valuations such that the tails "match" the database and then take the least upper bound of the heads. The BK-calculus is reviewed in Section 4. In our investigation, we view each query expression in COL and BK as a query with a special predicate \( \text{ans} \) for output.

"Domain Turing machines" (domTMs) were introduced in [34] and are a variant of Turing machines which is focused on database manipulations. Unlike conventional Turing machines, the tape alphabet of a domTM includes the (infinite) domain \( U \) while the transition function is still finitely expressed with the help of two "generic" symbols (\( \eta, \kappa \)). The main advantage of domTMs is that they can be used to simplify the proofs of the results of form "a query language expressed all queries computable

\[7\] In BK, this is based on the sub-object relationship instead of equality.
by Turing machine in $C$" where $C$ is a complexity class that is log-space or higher. We review domTMs and provide an example here; detailed discussions of domTMs can be found in [34].

In order to handle infinitely many symbols in the alphabet, each domTM also has a "register" that can store any single letter of the alphabet. Formally, a (deterministic) domain Turing machine (domTM) (relative to $U$) is a sextuple $M = (K, W, C, \delta, q_s, q_h)$ where

1. $K$ is a finite set of states.
2. $W$ is a finite set of working symbols. (In the current discussion we assume that $W$ includes the distinguished symbols ',', '(', ')', '[', ']' and '0' and '1' which are used for encoding input and output, and also $\beta$.)
3. $C \subseteq U$ is a finite set of constants.
4. $q_s \in K$ is the start state.
5. $q_h \in K$ is the unique halting state.
6. $\delta$ is the transition function

$$\delta(q, a, b) = (q', a', b', \text{dir})$$

only if $a = \eta$; $q \in \{a', b'\}$ only if $\eta \in \{a, b\}$; and $\kappa \in \{a', b'\}$ only if $\kappa \in \{a, b\}$.

$M$ has a two-way infinite tape and a register. An instantaneous description (ID) of $M$ is a quintuple $(q, a, \alpha, b, \beta)$ where $q$ is a state; $a \in W \cup U \cup \{\beta\}$ is the register content; $\alpha, \beta \in (W \cup U)^*$ and $b \in W \cup U$ such that the tape contents are $ab\beta$, where the tape head position is the specified occurrence of $b$. (We assume the usual restriction that neither the first symbol of $\alpha$ nor the last symbol of $\beta$ is $\beta$.) A transition value $\delta(q, a, b) = (q', a', b', \text{dir})$ is generic if $\eta \in \{a, b\}$. Intuitively, a generic transition value is used as a template for an infinite set of transition values which are formed by letting $\eta$ (and $\kappa$ if it occurs) range over (distinct) elements of $U - C$. At the beginning of a computation the register holds $\beta$. Under these provisions, a computation of $M$ is defined in the usual fashion.

**Example 2.10.** We define (parts of) a domTM $M$ which computes $\sigma_{1=3}(R)$ of a ternary relation $R$. This domTM will have no constants (i.e., $C = \emptyset$). The input of $M$ will be an enumeration of tuples in $R$. The transition function $\delta$ includes

1. $\delta(q_s, \beta, \emptyset) = (q_1, \beta, [\ , R])$
2. $\delta(q_1, \beta, ()) = (q_1, \beta, (, R))$
3. $\delta(q_1, \beta, \eta) = (q_2, \eta, \eta, R)$
4. $\delta(q_2, \eta, \eta) = (q_2, \eta, , R)$
5. $\delta(q_2, \eta, \kappa) = (q_3, \eta, \kappa, R)$
6. $\delta(q_2, \eta, \eta) = (q_3, \eta, \eta, R)$
Here transitions (1) and (2) get to the first atomic object of the input encoding. Transition (3) "remembers" that atomic object in the register. Transitions (4)–(7) skip over the second coordinate of the tuple. Transitions (8) and (9) compare the third coordinate of the tuple with the register contents; if they match transition (8) leaves the tuple unchanged, and if they do not match transition (9) changes the third coordinate to $v$ so that it can be deleted later in the computation. Both of these transitions also replace the register contents by $\beta$. Transition (10) reads over the ‘\(\beta\)', and transfers control back to transition (2). The end of the input is detected by transition (11), which turns control over the state $q_4$. Although the details are not included here, the computation from state $q_4$ erases the tuples marked by a $\$\$ in the third coordinate, and arranges the remaining tuples so that they are listed without separation.

The correspondence between domTMs and queries is not complete, since the input symbols are still ordered and the mappings computed by domTMs may depend on the input order. For this reason, we restrict to input-order independent domTMs that operate on enumerations of databases (tuples, relations) [34].

Proposition 2.11. [34]. The family of mappings computable by input-order independent domTMs is $\mathcal{C}$. The family of mappings computable by input-order independent domTMs in hyper-exponential time is $\mathcal{E}$.

3. Deductive languages and computable queries

We consider in this section two natural generalizations of COL [5] to recursively typed complex objects. The main result is that the two generalizations, with stratified and inflationary semantics respectively, are $\mathcal{C}$-equivalent. For completeness, it is shown here that under inflationary semantics, COL with non-recursive types is $\mathcal{E}$-equivalent. Thus, COL with the new (inflationary) semantics is equivalent to the original COL of [5] and a collection of other languages for complex objects listed in [27]. We also mention an enriched stratified semantics for the generalized COL which yields a language which is equivalent to the arithmetical hierarchy. These results also hold for the generalizations of the recursive language of [3] to include recursive types.

For clarity, we denote the original COL as $\text{c}_0\text{COL}$ and the extensions to r-types as $\text{r}_1\text{COL}$ in the remainder of the paper. We begin with a presentation of a straightforward generalization, $\text{r}_1\text{COL}$, of $\text{c}_0\text{COL}$ to r-types. Although a complete exposition of the original $\text{c}_0\text{COL}$ for non-recursive types is presented in [5], the presentation here
should suffice to introduce the reader to both forms of COL (with and without recursive types). The result on stratified \( r \)-COL for \( r \)-types is then presented. Inflationary versions of COL for both recursive and non-recursive types are then introduced, and the results about them presented.

\( r \)-COL is a DATALOG-like language for complex objects, which permits set-valued "data functions" in both input and computation, and which permits negation in the rule bodies. The syntax of \( r \)-COL uses:

- constants, i.e., elements of \( \mathbb{U} \);
- typed variables (typed by \( r \)-types);
- typed predicates for equality \( (=, \neq) \) for each pair of \( r \)-types \( \tau_1, \tau_2 \) such that \( \text{dom}(\tau_1) \cap \text{dom}(\tau_2) \neq \emptyset \) and set-membership \( (\in, \subseteq) \) for each pair of \( r \)-types \( \tau_1, \tau_2 \) such that there are \( u \in \text{dom}(\tau_1) \) and \( v \in \text{dom}(\tau_2) \) such that \( u \in v \);
- typed predicate symbols;
- typed function symbols of three kinds:
  - data functions (with domain \( \tau_1, \ldots, \tau_n \) and range \{\tau\}) for \( r \)-types \( \tau_1, \ldots, \tau_n, \tau \);
  - tuple constructors \( [t_1, \ldots, t_n] \);
  - set constructors \( \{t_1, \ldots, t_n\} \).

Terms are constants, variables, or data functions applied to terms (respecting the typing requirements). Following [5], we write \([t_1, \ldots, t_n]\) for \([t_1, \ldots, t_n]\) and \(\{t_1, \ldots, t_n\}\) for \(\{t_1, \ldots, t_n\}\).

There are three kinds of positive literals (again respecting the typing requirements): (i) \( p(t_1, \ldots, t_n) \) for a predicate symbol \( p \), or (ii) \( s = t_1, \ldots, t_n \), or (iii) \( s \in t_1, \ldots, t_n \). If \( \psi \) is a positive literal then \( \neg \psi \) is a negative literal. The subscripts indicating type are dropped if defined by the context; we use the abbreviations \( \neq \) and \( \in \). A term (literal) is ground if no variables occur in it. Two ground terms \([t_1, \ldots, t_n]\) and \(\{s_1, \ldots, s_m\}\) are identified if the sets \([t_1, \ldots, t_n]\) and \(\{s_1, \ldots, s_m\}\) are equal, or consist of pairwise identified terms. This identification is extended recursively when these terms occur as subterms.

An atom is a positive literal with form \( p(t_1, \ldots, t_n) \) or \( t \in F(t_1, \ldots, t_n) \) for some data function symbol \( F \). A rule is an expression of the form \( A \leftarrow L_1, \ldots, L_n \), where \( A \) is an atom and \( L_i \) is a literal for each \( 1 \leq i \leq n \). In this rule, \( A \) is the head and \( L_1, \ldots, L_n \) is the body. A \( (r \text{-COL, } co \text{-COL}) \) program is a finite set of rules.

Let \( P \) be an \( r \)-COL program. The set of predicate, data function, tuple constructor and set constructor symbols of \( P \) is denoted \( \mathbb{L}_P \). The (Herbrand) base of \( P \) is the set of all ground atoms constructed using constants (from \( \mathbb{U} \)) and \( \mathbb{L}_P \). An interpretation (over \( L_P \)) is a finite set of ground atoms constructed using arbitrary constants, the tuple and set constructors, elements in \( L_P \), and set-membership predicates. For an interpretation \( I \) and predicate symbol \( p \), \( I(p) \) denotes the set \( \{[o_1, \ldots, o_n] \mid p(o_1, \ldots, o_n) \in I\} \). (In \( co \)-COL, interpretations are also restricted to be finite. At the end of this section we briefly explore the implications of permitting infinite interpretations in the context of recursive types).

Let \( P \) be a fixed \( r \)-COL program. A ground substitution is a (partial) mapping \( \emptyset \) from variables to ground terms over \( L_P \) (respecting the typing of the variables). If

---

8 The reason for having two \( r \)-types is that it is possible for an object to be of two distinct \( r \)-types.

9 In \( co \)-COL, the type system is slightly more general than non-recursive types defined in Section 2. In particular, \( co \)-COL supports finitely heterogeneous set types \( r = \{\tau_1, \ldots, \tau_n\} \) with domain \( \mathbb{P}(\bigcup_{\tau_i} \text{dom}(\tau_i)) \).
I is an interpretation over \( L_P \), the valuation corresponding to \( \theta \) and \( I \) is a function \( \theta_I \) from the set of terms to the set of ground terms over \( L_P \) defined as follows.

- \( \theta_I \) is the identity for constants, and \( \theta_I(x) = \theta(x) \) for each variable in the domain of \( \theta \);
- \( \theta_I([t_1, \ldots, t_n]) = [\theta_I(t_1), \ldots, \theta_I(t_n)] \) and \( \theta_I([t_1, \ldots, t_n]) = \{ \theta_I(t_1), \ldots, \theta_I(t_n) \} \);
- \( \theta_I(F(t_1, \ldots, t_n)) = \{ t \mid t \in F(\theta_I(t_1), \ldots, \theta_I(t_n)) \} \in I \) (the right-hand side of the equality is viewed as a single term).

Valuations are extended to positive and negative literals in the natural manner. Finally, satisfaction (\( \models \)) and its negation (\( \not\models \)) or a rule or program by an interpretation are then defined in the classical way [44].

Let \( P \) be a program and \( I \) an interpretation. We use \( \text{adom}(I, P) \) to denote the set of constants occurring in \( I \) or \( P \). The immediate consequence operator \( T_P \) is defined for interpretations \( I \) over \( L_P \) by

\[
T_P(I) = \left\{ \theta(A) \mid A \in L_1, \ldots, L_n \in P, \theta \text{ is a ground substitution with range } \subseteq \text{adom}(I, P), \text{ and } \theta_I(L_i) \in I \text{ for each } 1 \leq i \leq n \right\}
\]

if this set is finite, and is undefined otherwise. Following \( \omega \text{COL} \), we include the restriction that the range of substitutions be contained in \( \text{adom}(I, P) \); this effectively yields the active-domain semantics – \( \text{adom}(o) \subseteq \text{adom}(P, I) \) for all objects \( o \) constructed by \( P \) on \( I \). (As a comparison, \( T_P(I) \) is always defined in \( \omega \text{COL} \).)

We also define the powers of operator \( T_P \) on interpretation \( I \) by

\[
T_P \uparrow n (I) = I \quad \text{if } n = 0 \quad \text{and } \quad T_P \uparrow (n+1) (I) = T_P(T_P \uparrow n (I)) \cup (T_P \uparrow n (I))
\]

If \( T_P \uparrow n (I) \) is undefined for some \( n < \omega \), then \( T_P \uparrow m (I) \) is undefined for each \( m > n \), and so is \( T_P \uparrow \omega (I) \). (Note that in \( \omega \text{COL} \), all of these are defined and furthermore, for each interpretation \( I \) this sequence will converge after a finite number of steps.)

**Example 3.1.** Consider the program \( P_1 \) with the single rule

\[
p([x], y, [w, z]) \leftarrow p(x, [w, y, z])
\]

where \( x, y, z \) are of r-type \( [O] \), \( w \) of r-type \( U \), \( p \) a predicate symbol of r-type \( [O, O, O] \), and \( S \) is a constant. If \( I \) is an interpretation containing the single atom \( p([S], t, [S]) \), where \( t = [a_1, [a_2, \ldots, [a_n, S]] \ldots] \) is an encoded sequence, then \( T_{P_1} \uparrow j (I) \) is defined for each \( j \geq 0 \), and the sequence converges after \( n \) steps. Specifically, after \( n \) steps \( p \) will include the triple \( ([S], S, r^k) \), where \( r^k = [a_n, \ldots, [a_1, [S]] \ldots] \) and \( S = [S, [S, \ldots, [S, [S]] \ldots]] \) (with \( n \) \( S \)'s).

Let the program \( P_2 \) contain the following two rules:

\[
q([S], [S]) \leftarrow q([S], [x, y]) \leftarrow q(x, y)
\]

where \( q \) is of r-type \( [U, O] \), \( x \) of r-type \( U \), and \( y \) of r-type \( O \). Then for each interpretation \( I \) and each \( n \geq 0 \), \( T_{P_2} \uparrow (n+1) (I) \) is defined and properly contains \( T_{P_2} \uparrow (n) (I) \); and \( T_{P_2} \uparrow (\omega) (I) \) is undefined since \( P_2 \) has only infinite models.
Finally, suppose the program $P_3$ consists of the single rule "$t(x) \leftarrow x = x$" where both $x$ and $r$ are of $t$-type 0. Then on any interpretation $I$ with non-empty active domain, $T_r(I)$ is undefined.

We now introduce definitions concerning stratification. This follows the general approach pioneered in [4,52], and as relativized to complex objects in [5]. The defined symbol of an atom $p(t_1, \ldots, t_n)$ is $p$, and of atom $t \in F(t_1, \ldots, t_n)$ is $F$. The defined symbol of a rule is the defined symbol of the head of the rule. Each predicate or data function symbol occurring in a rule is called a determinant of the rule. An occurrence of a determinant predicate $p$ is partial if that occurrence arises in a positive literal. The occurrence of a determinant function $F$ in a positive literal $t \in F(t_1, \ldots, t_n)$ is also partial. A determinant is partial in a rule if all its occurrences are partial; a determinant is total in the rule otherwise.

Let $P$ be a program. For defined symbols $X$ and $Y$ of $P$, we write $X < Y$ if $X$ is a total determinant of $Y$ in some rule of $P$, and $X \leq Y$ if $X$ is a partial determinant of $Y$ in some rule of $P$ but not a total determinant of $Y$ in any rule of $P$. We then construct a marked, directed graph $G_P$ whose nodes are all defined symbols in $P$, with a directed edge from $X$ to $Y$ if $X < Y$ or $X \leq Y$. Additionally, mark the edge $(X, Y)$, if $X < Y$.

The program $P$ is stratified if there is no cycle in $G_P$ that has a marked edge. (The same definitions are used for $\mathrm{COL}$.)

Let $P$ be a $\mathrm{COL}$ program with defined symbols $Q$. It is shown in [5] that $P$ is stratified iff there is a partition $Q_1 \uplus \cdots \uplus Q_k$ of $Q$ such that
1. $X \in Q_i, X \leq Y \implies$ for some $j$, $i < j$ and $Y \in Q_j$, and
2. $X \in Q_i, X < Y \implies$ for some $j$, $i < j$ and $Y \in Q_j$.

Such a partition induces a partition $P_1 \uplus \cdots \uplus P_n$ of $P$, where $P_i$ contains all rules which define symbols in $Q_i$. Such a partition is called a stratification of $P$. It is straightforward to verify that these results generalize to recursively typed $\mathrm{COL}$.

Finally, suppose that $P$ is a stratified $\mathrm{COL}$ program with stratification $P = P_1 \uplus \cdots \uplus P_n$. Let $I$ be an interpretation for $P$. For each $0 \leq j \leq n$, we define $I_j$ by

$I_0 = I$ and $I_{j+1} = T_{P_j} I_j$ for each $0 \leq j \leq n - 1$.

Then $I_n$ is the semantics of $P$ applied to $I$ under the stratification $P$. It is shown in [5] that the semantics of $P$ on an interpretation $I$ is the same regardless of the stratification used. We denote this by $P(I)$.

In generalizing this to $\mathrm{COL}$ for $t$-types, it may occur that $I_j$ is undefined at some stage; in this case the semantics $P(I)$ of $P$ on $I$ using that stratification is undefined. Using the same approach as in [5], it is straightforward to verify the following.

**Proposition 3.2.** Let $P$ be a stratified $\mathrm{COL}$ program, $P = P_1 \uplus \cdots \uplus P_n$ a stratification of $P$, and $I$ an interpretation over $L_P$. The semantics of $P$ on $I$ under $P$ is defined if and only if the semantics of $P$ on $I$ under any stratification of $P$ is defined. If the semantics of $P$ on $I$ under $P$ is defined, then that semantics is independent of the stratification used.

Let $D = \langle p_1; \tau_1, \ldots, p_n; \tau_n \rangle$ be a flat database schema. In this case we view $p_i$ as a unary predicate symbol of type $\tau_i$ for each $0 \leq i \leq n$. Also, let $\mathrm{ans}$ be a designated unary predicate symbol of type $\tau$. Suppose that $P$ is a stratified program such that $p_i$ is not a defined symbol of $P$ for each $1 \leq i \leq n$, and that $\mathrm{ans}$ is defined in $P$. Then
P can be viewed as defining a query function from D to \( \tau \), written \( P : D \rightarrow \tau \). In particular, let \( d = \langle p_1 : I_1, \ldots, p_n : I_n \rangle \) be an instance of D. The interpretation corresponding to \( d \), also denoted by \( d \), is \( \{ p_i(c_1, \ldots, c_k) \mid 1 \leq i \leq n, c_1, \ldots, c_k \in I_i \} \). If \( P[d] \) is defined, then \( P[d] \) denotes the instance \( P(d)(\text{ans}) \); otherwise \( P[d] \) is undefined.

We now present the following example to illustrate some r-types whose objects can “hold” configurations of domain Turing machines. We then provide the result showing that \( \mu \text{COL} \) with stratified semantics is \( \mathcal{C} \)-equivalent.

Example 3.3. Let \( M = (K, W, C, \delta, q_s, q_h) \) be a domTM. Without loss of generality, we assume that \( K \) and \( W \) are disjoint, \( (K \cup W) \cap U = \emptyset \), and \( \$ \not\in K \cup W \cup U \). We associate to each symbol \( a \in W \cup K \cup \{\$\} \) a distinct element \( a_U \in U \). Suppose that \( 1, 2 \) are two elements in \( U \). Now let the r-type \( T_{\text{sym}} = [U, U] \). We encode all symbols used by \( M \) into objects using the function \( \sim : W \cup K \cup \{\$\} \cup U \rightarrow \text{dom}(T_{\text{sym}}) \) (‘(a) is denoted as \( \tilde{a} \)) where \( \tilde{a} = [1, a] \) for each \( a \in U \) and \( \tilde{a} = [2, a_U] \) for each \( a \in K \cup W \cup \{\$\} \).

Finally, strings representing tape contents of \( M \) are appended by an explicit end symbol \( \$ \), and then encoded into objects of r-type \( T_{\text{tape}} = [T_{\text{sym}}, 0] \) by the function \( \sim^* : (W \cup U)^* \rightarrow \text{dom}(T_{\text{tape}}) \) (‘(a) is denoted as \( \hat{a} \)) defined as follows.

1. \( \hat{\varepsilon} = [\$, \$] \);
2. \( \hat{a} = [\tilde{a}, \hat{\varepsilon}] \) for \( a \in W \cup U \); and
3. \( \hat{a}x = [\tilde{a}, \hat{x}] \) for \( x \neq \varepsilon \).

Theorem 3.4. The family of stratified \( \mu \text{COL} \) programs is \( \mathcal{C} \)-equivalent.

Proof. It is easily verified that for each stratified \( \mu \text{COL} \) program \( P : D \rightarrow \tau \) with flat input and output, the mapping \( P[\cdot] \) is in \( \mathcal{C} \) (this remains true if the flat assumption is removed). To demonstrate the converse, we describe how a stratified \( \mu \text{COL} \) program can simulate an arbitrary domTM. This simulation has a structure similar to the simulation of domTMs by the algebraic language for recursive types used in [34].

Let \( M = (K, W, C, \delta, q_s, q_h) \) be a domTM which computes a query function \( f : D \rightarrow \tau \), where \( D = \langle p_1 : \tau_1, \ldots, p_n : \tau_n \rangle \), and let \( \text{ans} \) be a predicate symbol of flat r-type \( \tau \). Let \( \$ \) be a symbol not in \( W \cup U \), and recall the encoding functions \( \sim : (K \cup U) \cup \{\$\} \cup U \rightarrow \text{dom}(T_{\text{sym}}) \) and \( \sim^* : (W \cup U)^* \rightarrow \text{dom}(T_{\text{tape}}) \) in Example 3.3.

Define \( T_D = [T_{\text{sym}}, T_{\text{sym}}, T_{\text{tape}}, T_{\text{sym}}, T_{\text{tape}}] \).

We now define a stratified \( \mu \text{COL} \) program \( P_M \) which simulates \( M \). The program \( P_M \) will be the union of the following three parts:

1. \( P_{\text{in}} \), which transforms the input instance \( d \) into a family of enumerations \( \{e\}_{e \in e} \), each of which can be used as input for \( M \);
2. \( P_{\text{simu}} \), which simulates individual steps of \( M \) (simultaneously for each enumeration in \( e \)); and
3. \( P_{\text{out}} \), which transforms the output of \( M \) (if any) back into an instance of \( \tau \) as a result of \( f(d) \).

We describe each of these components in turn. To provide the intuition of the construction, suppose that \( d = \langle p_1 : I_1, \ldots, p_n : I_n \rangle \) is an input instance of the program \( P_M \).

\( P_{\text{in}} \) assigns to a unary predicate \( \text{ENC} \) of type \( T_{\text{tape}} \) all encodings \( \hat{e} \) of enumerations \( e \) of \( d \). This is accomplished by first making encodings for each of the input relations separately, and then concatenating them. For the first part, suppose \( \text{ENC}_j \) (\( 1 \leq j \leq n \))
is a unary predicate of type $T_{tape}$. Now for each $1 \leq j \leq n$, we first build the set of all
tuples in relation $I_j$ of $d$ using the rule

$$[x_1, \ldots, x_k] \in INPUT_j \leftarrow p_j([x_1, \ldots, x_k])$$

where $k$ is the arity of $p_j$, and $INPUT_j$ is a 0-ary data function with range r-type $U^k$. We now use a binary predicate $INC_j$ (for incremental) of r-type $[\{ U^k \}, O]$ to subtract
individual members from $I_j$ and insert them into a list.

$$INC_j(INPUT_j) \leftarrow$$

$$INC_j(minus^k(X, [z_1, \ldots, z_k]), ([z_1, \ldots, z_k], y)) \leftarrow INC_j(X, y), [z_1, \ldots, z_k] \in X$$

where the binary data function $minus^k$ from $\{ U^k \} \times \{ \tilde{U}^k \}$ to $\{ \tilde{U}^k \}$ is defined by the following rule

$$s \in minus^k(X, t) \leftarrow s \in X, s \neq t$$

and $\cdot(([z_1, \ldots, z_k], y) = ([1], [1, z_1], \ldots, [1, z_k], y) \ldots \ldots].$ Finally, we include

$$ENC_j(([1], [x])) \leftarrow INC_j(\{ }, x)$$

To concatenate the encoded enumerations held in the unary relations $ENC_j$ we first use the following rules for computing the reverse of the enumerations in $ENC_j$ for each $1 \leq j \leq n - 1$

$$REV_j(x, \tilde{e}) \leftarrow ENC_j(x)$$

$$REV_j(x, [u, y]) \leftarrow REV_j([u, x], y), \ u \neq \tilde{S}$$

$$ENC^\tilde{R}(y) \leftarrow REV_j(\tilde{e}, y)$$

where $u$ has r-type $[U, U]$, and $x, y$ have r-type $O$. (Since $\tilde{e} = [\tilde{S}, \tilde{S}]$, the condition $u \neq \tilde{S}$ in the second rule prevents it from being applicable when the third rule is.) Beginning
with the contents of $ENC_n$, we prepend enumerations from $ENC_{n-1}$ through $ENC_1$

$$BUILDING_j(x, y) \leftarrow BUILT_j(y), ENC^\tilde{R}(x)$$

The final encoding is now obtained by the rule "$ENC(x) \leftarrow BUILT_j(x)$".

Next we turn to $P_{simu}$. The central predicate here is $ID$ of r-type $T_{ID}$; $P_{simu}$ will be
constructed so that $ID(\tilde{q}, \tilde{a}, \tilde{b}, \tilde{e}, \tilde{R})$ holds iff $(q, a, x, b, \beta)$ is an instantaneous
description reached by $M$ when started on some encoding $e$ in $e$. The predicate $ID$ is initialized by

$$ID(\tilde{q}, \tilde{a}, \tilde{b}, \tilde{e}, [1], z) \leftarrow ENC(([1], z))$$

For each move of $M$ with form $\delta(q, a, b) = (q_1, a_1, b_1, -)$ we include

$$ID(\tilde{q}, \tilde{a}, [1], [1], [1], \tilde{b}, y) \leftarrow ID(\tilde{q}, \tilde{a}, \tilde{b}, y)$$

For each move of $M$ with form $\delta(q, \eta, \kappa) = (q_1, \kappa, a, -)$ we include

$$ID(\tilde{q}, [1], [1], [1], \tilde{a}, y) \leftarrow ID(\tilde{q}, [1], \tilde{a}, [1], [1], y), \ u \neq v, u \neq c_1, \ldots, u \neq c_m, \ v \neq c_1, \ldots, v \neq c_m$$
where \( u, v \) have \( r \)-type \( U \) and \( C = \{ c_1, \ldots, c_m \} \) is the set of constant used by the domTM \( M \).

For each move of form \( \delta(q, \eta, a) = (q_1, a_1, \eta, R) \) we include for each \( d \in W - \{ \$ \} \) the rule

\[
ID(\bar{q}_1, \bar{a}_1, [[1, u], x], \bar{d}, y) \leftarrow ID(\bar{q}, [1, u], x, \bar{a}, [\bar{d}, y]), \quad u \neq c_1, \ldots, u \neq c_m
\]

and the following two rules:

\[
ID(\bar{q}_1, \bar{a}_1, [[1, u], x], [1, v], y) \leftarrow ID(\bar{q}, [1, u], x, \bar{a}, [[1, v], y]), \quad u \neq c_1, \ldots, u \neq c_m
\]

\[
ID(\bar{q}_1, \bar{a}_1, [[1, u], x], \bar{\delta}, \bar{e}) \leftarrow ID(\bar{q}, [1, u], x, \bar{a}, \bar{e}), \quad u \neq c_1, \ldots, u \neq c_m
\]

Here the first deals with the situation that the head moves to the position that has a constant in \( U \) while the second simulates the creation by \( M \) of a new blank symbol because it is at the right end of the tape. Analogous rules are included for other forms of right- and left-moving transitions.

Finally, we include the rule

\[
M\text{-OUT}(\bar{1}, \bar{z}) \leftarrow ID(\bar{q}_b, u, \bar{c}, [\bar{1}, \bar{z}])
\]

It is straightforward to verify that if \( A4 \) halts on some enumeration \( e \) of \( d \), then \( M\text{-OUT} \) will hold the set \( \{ \bar{e}_1 \mid e_1 \text{ is the output of } M \text{ for some encoding } e \text{ of } d \} \), and otherwise this strata of \( \& \) will have undefined value on input \( I \).

We conclude with a description of \( P_{out} \). For this we need to extract the tuples encoded by the members of \( M\text{-OUT} \) and put them into a set of type \( \{ U^l \} \) (where \( l \) is the arity of \( \tau \)). To this end we include the rules

\[
DEC(x, \{ \} ) \leftarrow M\text{-OUT}(\bar{1}, x)
\]

\[
DEC(x, plus'(Y, [z_1, \ldots, z_l])) \leftarrow DEC([], [[1, z_1], [], \ldots, [1, z_l], [], x] \ldots]], Y)
\]

\[
y \in plus'(Y, t) \leftarrow y \in Y
\]

\[
t \in plus'(Y, t) \leftarrow
\]

To complete the query, we include the rule "\( ans(y) \leftarrow DEC(\bar{e}, Y), y \in Y \)" to produce the query answer.

It can be verified that (1) the program \( P_M \) is stratified and (2) for each database \( d \), \( P_M[d] \) is defined and equals \( f(I) \) if \( M \) halts on enumeration of \( I \), and \( P_M[d] \) is undefined otherwise. This concludes the proof. \( \square \)

The "inflationary" semantics for DATALOG was introduced in [39,9]. Under this semantics, the operator \( T_p \) is repeatedly applied to an interpretation until convergence (if ever). The result will be a model of \( P \), but may not be a minimal model of \( P \). It is known that inflationary DATALOG is strictly more expressive than stratified DATALOG [38,39,9]. After defining the inflationary semantics for \( \_col \) and \( \_COL \) we demonstrate that inflationary COL has the same expressive power as stratified COL in both non-recursive and recursive type cases.

We begin with recursively typed \( _r \)COL. For an arbitrary \( _r \)COL program \( P \) and interpretation \( I \) over \( L_p \), the inflationary semantics of \( P \) on \( I \), is \( P^{infl}(I) = T_p \uparrow_{\_r \_COL}(I) \), if defined. (This is always defined in the case of \( \_CO \) COL.) As with strat-
ified COL, if \( D \) is a flat database schema and \( ans \) a designated predicate symbol of flat \( r \)-type \( \tau \), and if \( P : D \rightarrow \tau \), then for instances \( d \) of \( D \) we denote the output of \( P^{\infty}[d] \) on \( ans \) by \( P^{\infty}[d] \).

**Example 3.5.** Let \( p, ans \) be unary predicate symbols of \( r \)-type \( U^k \) for some \( k > 1 \), and let \( P \) be the program

\[
INPUT(x) \leftarrow p(x) \\
ans(x) \leftarrow \neg INPUT(X)
\]

where \( INPUT \) is a predicate of \( r \)-type \( U^k \). Then \( P \) is stratified, and on input \( d = \langle p : I \rangle \) we have \( P[d] = adom(d)^k - I \). On the other hand, under the inflationary semantics \( P^{\infty}[d] = adom(d)^k \). This is because \( T_p(d)(ans) = adom(d)^k \); and so \( T_p \uparrow_1 (D)(ans) \geq T_p \uparrow_1 (d)(ans) = adom(d)^k \).

**Example 3.6.** Let \( CTR \) be a binary predicate symbol of type \( [T \text{sym}, T \text{tape}] \), and suppose that the program \( P \) has the following two rules defining \( CTR \):

\[
CTR(\hat{e}) \leftarrow \\
CTR([\hat{S}, \hat{x}]) \leftarrow CTR(x)
\]

It is easily verified that with any input interpretation \( I \), the presence of these rules implies for each \( i \geq 0 \) that \( [T_p \uparrow_1 (I)][CTR] = \{ \hat{S} \mid 1 \leq j \leq i \} \) (if defined). Suppose now that the only rule of \( P \) which defines the symbol \( X \) is

\[
A \leftarrow L_1, \ldots, L_m, CTR(\hat{S}^{\prime})
\]

Then \( [T_p \uparrow_j (I)][X] \) is empty for each \( j < m \). Thus, the relation \( CTR \) can be used to delay the construction of data functions and relations for a bounded number of iterations of the application of \( T_p \). (This is reminiscent of a technique used in [8].)

As a concrete example, let \( P_1 \) be the union of the two rules for \( CTR \) given above and

\[
INPUT(x) \leftarrow p(x) \\
ans(x) \leftarrow \neg INPUT(x), CTR(\hat{S})
\]

i.e., the result of adding a \( CTR \) condition to the second rule of the program \( P \) of Example 3.5. Then on all inputs \( d \) we have \( P[d] = P^{\infty}[d] \).

Example 3.6 exhibits a technique to delay the evaluation of rules for a fixed number of steps. One can also apply the timestamp method described in [7] to delay rule evaluation for a variable number of steps (depending on the results at previous steps). This method is used in [7] to prove that each stratified DATALOG query can be simulated by an inflationary DATALOG program. This proof can be easily extended to establish the following (proof omitted).

**Lemma 3.7.** The stratified \( \pi \)COL is no more expressive than (\( \sqsubseteq \)) the inflationary \( \pi \)COL.

**Theorem 3.8.** The family of \( \pi \)COL programs interpreted using inflationary semantics is \( \phi \)-equivalent.
Proof. It is immediate that all mappings with flat input and output defined by inflationary $\pi\text{COL}$ are in $\text{\&}$ (this again holds without the flat assumption). For the converse, since each query function in $\text{\&}$ can be realized by a stratified $\pi\text{COL}$, it can also be realized by an inflationary $\pi\text{COL}$ program by Lemma 3.7. □

For completeness, we consider inflationary $\cong\text{COL}$ (i.e., inflationary $\pi\text{COL}$ without recursive types). Clearly, the general technique of simulating stratified semantics in an inflationary language applies here and the containment similar to Lemma 3.7 can be established. Since $\cong\text{COL}$ is $\delta$-equivalent [33,27,28], we have the following proposition.

**Proposition 3.9.** The family of $\cong\text{COL}$ programs under inflationary semantics is $\delta$-equivalent. Thus, $\cong\text{COL}$ under inflationary semantics and under stratified semantics are equivalent.

**Remark 3.10.** In the definitions and results of this section we followed the original definition of $\pi\text{COL}$ by permitting only finite interpretations. We consider briefly here the implications of permitting infinite interpretations in the recursively typed setting. (Infinite objects are still prohibited). In the case of stratified semantics, this yields the expressive power of the arithmetical hierarchy [34]. A proof of this can be modeled after the analogous result for stratified logic programs in [2]. Characterization of the expressive power of inflationary $\pi\text{COL}$ in this context remains open.

### 4. The power of the BK-calculus

In this section we analyze the power of the BK-calculus [13]. The BK-calculus is a deductive language, and was one of the first languages for recursive types (more specifically, it uses untyped sets). It is distinguished from most other deductive languages by its use of a "sub-object" relationship rather than the equality in the application of program rules. As a result, BK-programs define monotonic database mappings (in the lattice defined by the sub-object relation) even though there are no restrictions such as stratification on the use of sets.

Two main results are presented in this section. The first (Theorem 4.4) focuses on BK-programs which map flat input to flat output; under a nominal technical condition, it is shown that such programs have expressive power corresponding to a natural variant of the conjunctive queries [18]. Significantly, because of the use of the sub-object relationship, this variant cannot compute the natural join (Corollary 4.10). This raises the natural question of the richness of the BK-calculus on unrestricted inputs (i.e., not necessarily flat). Theorem 4.21 shows that BK-programs can simulate Turing machines when given suitably encoded inputs. Consequently, BK-calculus has unbounded data complexity and it is undecidable if a given BK-program is defined (producing output) over a given database.

To begin this section we briefly review the core concepts of the BK-calculus. We closely follow the development of [13], although some of our notation is chosen to parallel the notation used for COL in Section 3. We then introduce the variant of conjunctive queries, and state the first theorem. The key concept of "BK-homo-
morphism” is introduced and studied, ultimately leading to the proof of the first theorem. The second theorem then follows.

In the BK-calculus attributes names are associated with tuple components. We let $A$ be an infinite set of attribute names, which is disjoint from $U$. The family of (not-necessarily-reduced) (BK-)objects is defined recursively as follows.

1. Each element of $U$ is an atomic object.
2. $\perp$ and $T$ are objects.
3. If $o_1, \ldots, o_n$ are objects and $A_1, \ldots, A_n$ are distinct attributes ($n \geq 0$) then $[A_1; o_1, \ldots, A_n; o_n]$ is a (tuple) object.
4. If $o_1, \ldots, o_n$ are objects ($n \geq 0$) then $\{o_1, \ldots, o_n\}$ is a (set) object.

A tuple object $[A_1; o_1, \ldots, A_n; o_n]$ is identified with $[A_1; o_1, \ldots, A_{i-1}; o_{i-1}, A_i; o_{i+1}, \ldots, A_n; o_n]$. A set object $o$ that includes the member $\perp$ is identified with the set $o - \{\perp\}$. If $T$ occurs within an object, the object is identified with $T$. For a tuple object $o = [A_1; o_1, \ldots, A_n; o_n]$, $o.A_i$ denotes $o_i$ for $1 \leq i \leq n$.

The depth of an object $o$, denoted $\text{depth}(o)$, is defined recursively so that $\text{depth}(\perp) = 1$; the depth of all atomic objects is 1; $\text{depth}(\{\}) = \text{depth}(\{\}) = 2$; the depth of $T$ is infinite; the depth of a set or tuple constructed from $o_1, \ldots, o_n$ is $1 + \max\{\text{depth}(o_i) \mid 1 \leq i \leq n\}$. The sub-object relationship $\leq$ is defined recursively so that

1. $\perp$ is $\leq$ every object and every object is $\leq T$;
2. $o$ is a sub-object of $o$ for each object $o$;
3. if $o$ and $o'$ are tuple objects and if $o.A \leq o'.A$ for each attribute $A \in A$, then $o \leq o'$; and
4. if $o$ and $o'$ are set objects and if for each $o_1 \in o$ there is an $o'_1 \in o'$ such that $o_1 \leq o'_1$, then $o \leq o'$.

We write $o \equiv o'$ if both $o \leq o'$ and $o' \leq o$. It is shown in [13] that $\leq$ is reflexive and transitive, but not anti-symmetric. To resolve this, attention is focused on reduced objects, which are recursively defined by:

1. $\perp$, $T$, and atomic objects are reduced.
2. A tuple $o$ is reduced iff $o.A$ is reduced for each $A \in A$.
3. A set $o$ is reduced iff each of its elements is reduced, and it does not contain two distinct objects $o_1$ and $o_2$ such that $o_1 \leq o_2$.

For an object $o$ the reduced version of $o$ is denoted by $\text{red}(o)$.

(Finitary) union ($\cup$) and intersection ($\cap$) of reduced objects is defined recursively by:

1. $T \cup o = T$, $\perp \cup o = T \cap o = o$, and $\perp \cap o = \perp$;
2. for atomic objects $o_1$ and $o_2$, $o_1 \cup o_2 = o_1$ if $o_1 = o_2$, and $T$ otherwise; $o_1 \cap o_2 = o_1$ if $o_1 = o_2$, and $\perp$ otherwise;
3. for tuple objects $o_1$ and $o_2$, $o_1 \cup o_2 = T$ if there is some $A \in A$ such that $o.A \cup o_2.A = T$; otherwise $o_1 \cup o_2 = o$ such that $o.A = o_1.A \cup o_2.A$ for each $A \in A$; $o_1 \cap o_2 = o$ such that $o.A = o_1.A \cap o_2.A$ for each $A \in A$; and
4. for set objects $o_1$ and $o_2$, $o_1 \cup o_2 = \text{red}(o)$ where $o$ is the (set-)union of $o_1$ and $o_2$; $o_1 \cap o_2 = \text{red}(o)$ where $o = \{o'_1 \cap o'_2 \mid o'_1 \in o_1$ and $o'_2 \in o_2\}$.

Note that for a finite set $S$ of objects, $\bigcup S$ is defined as $\text{red}(\bigcup S) = \bigcup\{\text{red}(s) \mid s \in S\}$; and that the set $\{|A; a, B; b\} \cup \{|A; \perp, B; b\}$ is reduced if $a$ and $b$ are distinct atomic objects. It is shown in [13] that the family of reduced objects along with $\leq, \cap, \cup$ forms a lattice.

Well-formed formulas (wffs) are defined recursively in the following.
1. A variable is a wff,
2. an atomic object is a wff,
3. if \( \phi_1, \ldots, \phi_n \) are wffs and \( A_1, \ldots, A_n \) are distinct attributes \((n \geq 0)\), then
   \[
   [A_1 : \phi_1, \ldots, A_n : \phi_n]
   \]
   is a wff, and
4. if \( \phi_1, \ldots, \phi_n \) are wffs \((n \geq 0)\) are wffs then \( \{ \phi_1, \ldots, \phi_n \} \) is a wff.

For each wff \( \phi \), let \( \text{var}(\phi) \) denote the set of all variables occurring in \( \phi \).

A substitution for a wff \( \phi \) is a function \( \sigma \) from \( \text{var}(\phi) \) to the set of reduced objects.

In this case, \( \sigma(\phi) \) is defined in the natural manner. A \((\text{BK})\)-rule in an expression \( \phi \leftarrow \phi' \) where \( \phi, \phi' \) are wffs and \( \text{var}(\phi) \subseteq \text{var}(\phi') \). For an object \( o \), the result of applying rule \( \gamma = \phi \leftarrow \phi' \) to \( o \) is defined as: \( \gamma(o) = \bigcup \{ \sigma(\phi) \mid \sigma \text{ is a substitution for } \phi' \text{ and } \sigma(\phi') \leq o \} \).

A BK-program is a finite set of rules. For a BK-program \( Q \) and object \( o \), \( Q(o) \) denotes a “single” application of the rules in \( Q \) to \( o \): \( Q(o) = \bigcup \{ \gamma(o) \mid \gamma \in Q \} \). The closure of \( Q \) on \( o \), denoted \( Q^*(o) \) is defined to be the least object \( o' \geq o \) such that \( Q(o') \leq o' \), if any such object \( o' \) exists.

In analogy with logic programming and \( \infty \text{COL} \) we define the immediate consequence operator \( T^i \) for a BK-program \( Q \) so that \( T^i(o) = Q(o) \) for each object \( o \).

The powers of \( T^i \) are defined by:

\[
\begin{align*}
T^0(o) & = o, \\
T^i(o) & = T(T^i-1(o)) \cup T^i-1(o), \quad \text{if defined}, \\
T^i_n(o) & = T^i_{n+1}(o) & \text{where } n \text{ is least such that } T^i_n(o) = T^i_{n+1}(o) \quad \text{if one exists.}
\end{align*}
\]

If \( T^i_n(o) \) is defined and \( n \) is least such that \( T^i_n(o) = T^i_{n+1}(o) \) then we say that the \( T^i \) \((o) \) sequence converges at \( n \). A result \(^{10}\) in [13] implies that if \( Q^*(o) \) is defined, then \( T^i_n(o) \) is defined, and \( Q^*(o) = T^i_n(o) \).

Intuitively, there are two possible behaviors of the \( T^i \) \((o) \) sequence in the case where it does not converge. First, the sequence of objects \( o_i = T^i(o) \) \((i \geq 0)\) might have no repeats, each one dominating the previous one (see Example 4.6 of [13]). Second, there may be some \( i \) such that \( T^i(o) = \top \) (see Example 4.15(b)). In that case, \( Q(T^i(o)) = Q(\top) \) should be the union of the set \( \{ \sigma(\phi) \mid \phi \leftarrow \phi' \text{ is a rule in } Q \text{ and } \sigma(\phi') \leq \top \} \). Since all objects are \( \leq \top \) this set is infinite, and its union is not defined. (Note that since all rules \( \phi \leftarrow \phi' \) satisfy \( \text{var}(\phi) \subseteq \text{var}(\phi') \), the object \( o \) cannot yield an infinite set in this manner unless \( o_i = \top \).)

We now turn to analyzing the expressive power of BK in the context of flat input and output. To simulate relations we use the following. An attributed relation schema is an expression \( R : [A_1, \ldots, A_n] \), where the \( A_i \)'s are distinct attribute names. If the sequence of labels is understood from the context, this schema is denoted simply by \( R \).

A tuple object \( t = [b_1; b_2; \ldots, b_n] \) has type \( R \) if \( b_i \)’s are atomic objects and \( \{b_1, \ldots, b_n\} \subseteq \{A_1, \ldots, A_n\} \). In this case, \( t \) is viewed to have the null-value on each attribute in \( \{A_1, \ldots, A_n\} - \{b_1, \ldots, b_n\} \). A \((\text{BK})\)-instance of \( R \) is a set of tuples of type \( R \). Finally, if the BK object \( o = [\ldots, R : \{t_1, \ldots, t_n\}, \ldots] \) where \( t_i \)'s are tuples of type \( R \), the \( o(R) \) denotes the instance \( \{t_1, \ldots, t_n\} \) of \( R \).

\(^{10}\) In [13] the series is specified as \( o_1 = Q(o_0), o_2 = Q(o_1), \ldots ; \) this seems to be in error, because it is not necessarily the case that \( Q(o_i) \geq o_i \) for \( i \geq 1 \).
Let \( Q \) be a BK-program that operates on tuple objects, let \( D = (R_1 : [A_1', \ldots, A_{m_1}], \ldots, R_n : [A_1^n, \ldots, A_{m_n}]) \) be a flat relational schema and \( R : [B_1, \ldots, B_m] \) a flat relation schema, where the attribute sets of different relations may overlap. We now view \( Q \) as a mapping from \( D \) to \( R \) as follows: An input instance for this mapping is an instance of the relational schema \( D \), i.e., a sequence \( d = (R_1 : I_1, \ldots, R_n : I_n) \), where \( I_j \) is an instance of \( R_j \) for \( 1 \leq j \leq n \). If \( Q'(d) \) is defined, then \( Q[d] \) denotes \( Q'(d)(R) \).

The cornerstone of the characterization of the expressive power of the BK-calculus on flat input and output is provided by the following variant of conjunctive queries.

**Definition 4.1.** A conjunctive query rule (CQ-rule) from \( D = (R_1 : [A_1', \ldots, A_{m_1}], \ldots, R_n : [A_1^n, \ldots, A_{m_n}]) \) to \( R : [B_1, \ldots, B_m] \) is a BK-rule of the following form:

\[
[R : \{[B_i_1 : x_1, \ldots, B_i_k : x_k]\}] \leftarrow [R_1 : \{t_1', \ldots, t_{k_1}\}, \ldots, R_n : \{t_n', \ldots, t_{k_n}\}],
\]

where \( x_j \)'s (\( 1 \leq j \leq k \)) are (not necessarily distinct) variables, for each \( 1 \leq l \leq n, k_i \geq 0 \) and \( t_j' \)'s are tuples over attributes \( A_1', \ldots, A_{m_1} \).

Note that if a program \( Q \) consists of CQ-rules from \( D \) to \( R \), then on a flat input \( d \), \( Q'(d) \) is always defined, and is obtained by a single (parallel) application of the rules in \( Q \) to \( d \). (That is, the \( T_0 \uparrow_1 (d) \) sequence converges at \( i = 1 \).)

The following example considers a particular CQ-rule, and shows the impact of the use of the sub-object relationship when applying it.

**Example 4.2.** Let \( D = (S : [A, B], T : [B, C]) \) and \( R : [A, C] \) be flat schemas, and consider a BK-program \( Q \) with the single rule

\[
[R : \{[A : x, C : z]\}] \leftarrow [S : \{[A : x, B : y]\}, T : \{[B : y, C : z]\}].
\]

Then on input \( d \) with no null values, \( Q'(d) = T_0 \uparrow_1 (d) \) is always defined. We now argue that (with a slight abuse of notation) \( Q[d] = \pi_A(d(S)) \times \pi_C(d(T)) \). To see that \( Q[d] \) contains the cross-product, suppose that \( [A : a, B : b] \in d(S) \) and \( [B : b', C : c] \in d(T) \). Let \( \sigma \) be an assignment satisfying \( \sigma(x) = a, \sigma(y) = y, \) and \( \sigma(z) = c \). Then \( \sigma([S : \{[A : x, B : y]\}], T : \{[B : y, C : z]\}]) \leq d \), and so \( \sigma([R : \{[A : x, C : z]\}]) = [R : \{[A : a, C : c]\}] \leq Q(d) \). Given the form of \( Q \), this implies that \( [A : a, C : c] \in Q[d] \) as desired.

To see that the cross-product contains \( Q[d] \) suppose that \( r \in Q[d] \). It follows that there is some assignment \( \sigma \) such that (1) for some \( s \in d(S) \), \( \sigma([A : x, B : y]) \leq s \), (2) for some \( t \in d(T) \), \( \sigma([B : y, C : z]) \leq t \), and (3) \( r = \sigma([A : x, C : z]) \). Let \( r' = [A : s[A], C : t[C]] \), and \( \sigma' \) be defined so that \( \sigma'(x) = s[A], \sigma'(y) = \sigma(y), \) and \( \sigma'(z) = t[C] \). Then \( r \leq r' = \sigma'([A : x, C : z]) \in Q[d] \). Since \( Q[d] \) is reduced, \( r = r' \) and, in particular, \( r \) is in the cross-product as desired. We further note that if input \( d \) has null values, then \( Q[d] \) is the reduced object equivalent to \( \pi_A(d(S)) \times \pi_C(d(T)) \).

Speaking intuitively, the above example illustrates that using the naive approach, BK is unable to "match" pairs of atomic objects based on the equality of related atomic objects. As we shall see in Corollary 4.10 below, there is no BK-program that computes the natural join, and hence, no program that can do this kind of matching.
On the other hand, Theorem 4.21 illustrates a kind of encoding by which BK can implement matching of certain structured objects.

To provide additional intuition concerning CQ-rules we include the following example.

**Example 4.3.** Let $D = \langle S : [A, B], T : [B, C] \rangle$ and $R : [A, C]$ be flat schemas, $\gamma_1$ the rule in Example 4.2 and

$$
\gamma_2 : \langle R : ([A : x, B : y, C : z]) \rangle \prec \langle S : ([A : x, B : y]), T[B : y, C : z] \rangle
$$

Then for all inputs $d$, $\gamma_1(d) \leq \gamma_2(d)$.

As will be seen below (Proposition 4.14), it is undecidable whether a BK-program is defined on a given input. For this reason, our characterization of BK-programs with flat input and output is restricted to those programs which are defined on all flat inputs. A generalization of this result to programs with some undefined output is presented in Corollary 4.12.

**Theorem 4.4.** Let $Q$ be a BK-program from $D = \langle R_1 : [A_1, \ldots, A^n_1], \ldots, R_n : [A^n_1, \ldots, A^n_m] \rangle$ to $R : [B_1, \ldots, B_m]$ that is defined on all inputs without null values. Then there is a BK-program $Q'$ consisting entirely of CQ-rules from $D$ to $R$ such that on each input instance $d$ (possibly with null values), $Q'[d] = Q[d]$.

The basic notion used to prove this theorem is that of “BK-homomorphism”. We introduce that notion now, and prove a series of lemmas about the interaction of homomorphisms and the application of BK programs. After that, the formal proof of the theorem is presented.

**Definition 4.5.** A (BK-)homomorphism $\mu : U \rightarrow U \cup \{\bot\}$. The domain of $\mu$ is denoted by $\text{dom}(\mu)$. A homomorphism $\mu$ is extended to the set of (reduced and non-reduced) objects by setting $\mu(\bot) = \bot, \mu(\top) = \top$, and defining $\mu(o)$ in the natural recursive manner for each object $o$ satisfying $\text{adorn}(o) \subseteq \text{dom}(\mu)$. Finally, for an arbitrary object $o$ with $\text{adorn}(o) \subseteq \text{dom}(\mu)$, $\bar{\mu}(o)$ is defined to be the reduced object equivalent to $\mu(o)$.

The need for $\bar{\mu}$ is illustrated by the homomorphism $\mu$ defined to be the identity except that $\mu(c) = a$, and the object $o = \{\{a, b\}, \{c\}\}$; in this case $\mu(o) = \{\{a, b\}, \{a\}\}$ is not reduced, and $\bar{\mu}(o) = \{\{a, b\}\}$. It is easily verified that if $\mu$ and $\nu$ are homomorphisms and $o$ an arbitrary object, then $\bar{\mu}(\nu(o)) = \bar{\mu}(\nu(o)) = \bar{\nu}(\nu(o))$.

**Definition 4.6.** A homomorphism $\mu$ is projecting if for each $d \in U$, either $\mu(d) = d$ or $\mu(d) = \bot$.

Three straightforward lemmas about BK homomorphisms are now stated.

**Lemma 4.7.** Let $\mu$ be a homomorphism and $o$ and $o'$ be reduced objects. Then:

1. If $o \leq o'$ then $\bar{\mu}(o) \leq \bar{\mu}(o')$.
2. $\bar{\mu}(o \cup o') \nleq \bar{\mu}(o) \cup \bar{\mu}(o')$; furthermore, if $o \cup o' \nleq \top$ then $\bar{\mu}(o \cup o') = \bar{\mu}(o) \cup \bar{\mu}(o')$. 
3. \( \mu(o \cap o') \leq \mu(o) \cap \mu(o') \).
4. If \( \mu \) is projecting then \( \mu(o) \leq o \).

Proof. This lemma is demonstrated by a straightforward induction on the definitions of \( \leq \), union, and intersection of objects. We include the special case of \( o \cup o' \neq \top \) in item (2) because it will be used below. The analog for item (3) does not hold, because \( \bot \) can occur within a BK-object, whereas if \( T \) occurs within a BK-object \( o'' \) then \( o'' \) is equivalent to \( T \). □

If \( \mu(a) = \mu(b) = c \) then \( \mu([A:a] \cup [A:b]) = \top \) and \( \mu([A:a]) \cup \mu([A:b]) = [A:c] \). Also, \( \mu([A:a] \cap [A:b]) = \bot \) and \( \mu([A:a]) \cap \mu([A:b]) = [A:c] \).

Lemma 4.8. Let \( \mu \) be a homomorphism, \( Q \) a program, and \( o \) an object. Suppose further that \( \text{adom}(o) \subseteq \text{dom}(\mu) \) and \( Q(o) \) is defined but not equal to \( \top \). Then (a) \( \mu(Q(o)) \leq Q(\mu(o)) \). Furthermore, (b) if \( \mu \) is an isomorphism or projecting homomorphism then \( \mu(Q(o)) = Q(\mu(o)) \).

Proof. For part (a), let \( S = \{\sigma(\phi) | \phi \leftrightarrow \phi' \in Q \) and \( \sigma(\phi') \leq o\} \) and \( T = \{\tau(\phi) | \phi \leftrightarrow \phi' \in Q \) and \( \tau(\phi') \leq \mu(o)\} \). Then \( Q(o) = \bigcup S \) and \( \mathcal{O}(\mu(o)) = \bigcup T \), so it suffices to show that \( \mu(\bigcup S) \leq \bigcup T \). Since \( Q(o) \) is defined, \( S \) must be finite and by assumption \( \bigcup S \neq \top \). Thus, \( \mu(\bigcup S) \equiv \mu(\bigcup S) = \bigcup \{\mu(s) | s \in S\} \) by Lemma 4.4(2). Suppose now that \( o' \in \{\mu(s) | s \in S\} \), i.e., for some \( \sigma \cdot \phi' \in Q \) and some substitution \( \sigma, \sigma(\phi') \leq o \) and \( o' = \mu \cdot \sigma(\phi) \). Letting \( \tau = \mu \cdot \sigma \), item (1) of Lemma 4.7 implies that \( \mu(\bigcup S) \equiv \mu(o) \), whence \( o' = \tau(\phi) \in T \). This implies that \( \bigcup \{\mu(s) | s \in S\} \leq \bigcup T \), which yields the desired result.

For part (b), if \( \mu \) is an isomorphism and \( o \) a reduced object, then \( \mu(o) = \mu(o) \) and the result is easily established. Suppose now that \( \mu \) is a projecting homomorphism. Again let \( S = \{\sigma(\phi) | \phi \leftrightarrow \phi' \in Q \) and \( \sigma(\phi') \leq o\} \) and \( T = \{\tau(\phi) | \phi \leftrightarrow \phi' \in Q \) and \( \tau(\phi') \leq \mu(o)\} \). It suffices to show that \( \mu(\bigcup S) \geq \bigcup T \). To this end, let \( t \in T \). Then for some substitution \( \tau, \tau(\phi') \leq \mu(o) \) and \( t = \tau(\phi) \). Since \( \mu(o) \leq o \) (by Lemma 4.7(4)), \( \tau(\phi') \leq o \), and so \( \tau(\phi) \in S \). Since \( \text{var}(\phi) \leq \text{var}(\phi') \), \( \text{adom}(\tau(\phi)) \subseteq \text{adom}(\tau(\phi')) \). Since \( \tau(\phi') \leq \mu(o) \), \( \{d \in U | \mu(d) = d\} \supseteq \text{adom}(\mu(o)) \supseteq \text{adom}(\tau(\phi')) \). It follows that \( \mu(\tau(\phi)) = \tau(\phi) \). Thus,

\[
\mu(\bigcup S) = \bigcup \{\mu \cdot \sigma(\phi') | \sigma(\phi') \leq o\} \quad \text{by Lemma 4.7(2)}
\supseteq \bigcup \{\tau(\phi) | \tau(\phi') \leq \mu(o)\} \quad \text{observations above}
\]

\[
= \bigcup T
\]

as desired. □

We present an example for which \( \mu(Q(o)) \neq Q(\mu(o)) \). Let \( Q \) be the program containing the only rule \([R : \{[C:x]\}] \leftarrow [R' : \{[A:x,B:B]\}] \); let \( \mu(a) = \mu(b) = c \); and let \( o = [R' : \{[A:a,B:B]\}] \). Then \( \mu(R : \{[C: \bot]\}) = \mu(Q(o)) \leq \mu(Q(\mu(o)) = Q([R' : \{[A:c,B:B]\}]) = R : \{[C:c]\}] \), which are not equal.

Finally, we generalize the above lemma to closures of BK-programs.

Lemma 4.9. Let \( Q \) be a BK-program.
1. If both \( Q^*(o) \) and \( Q^*(\mu(o)) \) are defined, then \( \mu(Q^*(o)) \leq Q^*(\mu(o)) \).
2. If \( Q' \langle o \rangle \) is defined and \( \mu \) is an isomorphism or projecting homomorphism, then \( Q' \langle \mu(o) \rangle \) is defined and \( \mu(Q' \langle o \rangle) = Q' \langle \mu(o) \rangle \).

**Proof.** Consider part (a). Let \( o \) be fixed and suppose that both \( Q' \langle o \rangle \) and \( Q' \langle \mu(o) \rangle \) are defined. Then \( Q' \langle o \rangle = T_Q \upharpoonright_n (o) \) and \( Q' \langle \mu(o) \rangle = T_Q \upharpoonright_m (\mu(o)) \) for some \( n, m \). A straightforward induction using Lemma 4.8 implies that \( \mu(T_Q \upharpoonright_i (o)) \subseteq T_Q \upharpoonright_i (\mu(o)) \) for each \( i \), which yields the desired result.

Part (b) is trivial for the case where \( \mu \) is an isomorphism. Suppose now that \( \mu \) is a projecting homomorphism and that \( Q' \langle o \rangle \) is defined. By Lemma 4.7(4), \( \mu(o) \equiv o \). By assumption, the \( T_Q \upharpoonright_i (o) \) sequence converges at some \( n \geq o \). A straightforward induction using Lemma 4.8 (b) shows that \( \mu(T_Q \upharpoonright_i (o)) = T_Q \upharpoonright_i (\mu(o)) \) for each \( i \geq 0 \). It follows that the \( T_Q \upharpoonright_i (\mu(o)) \) sequence converges after \( n \) steps, \( Q' \langle \mu(o) \rangle \) is defined, and \( Q' \langle \mu(o) \rangle = \mu(Q' \langle o \rangle) \). □

**Corollary 4.10.** Let \( D = \langle S : [A, B], T : [B, C] \rangle \) and \( R : [A, C] \) be flat schemas. There is no BK-program from \( D \) to \( R \) which computes the natural join.

**Proof.** Suppose that \( Q \) is a BK-program from \( D \) to \( R \) which computes the natural join on \( D \). Let \( d \) and \( d' \) be the instances with

\[
\begin{align*}
d(S) &= \{ [A:a, B:b] \}, & d'(S) &= \{ [A:a, B:e] \}, \\
d(T) &= \{ [B:b, C:c] \}, & d'(T) &= \{ [B:e', C:c] \},
\end{align*}
\]

where \( a, b, c, e, e' \) are distinct atoms. By assumption, \( t = [A:a, C:c] \in Q[d] \). Let \( \mu \) be the homomorphism such that \( \mu(b) = \bot \) and is the identity elsewhere. Note that \( \mu(d) = \mu(d') \). Also, \( \mu \) is projecting and so by Lemma 4.9(2), \( \mu(Q[d]) = Q[\mu(d)] \) and \( \mu(Q[d']) = Q[\mu(d')] \). Thus, \( [A:a, C:c] = \mu([A:a, C:c]) \in Q[\mu(d)] = Q[\mu(d')] = \mu(Q[d']) \).

Thus \( [A:a, C:c] \in Q[d'] \neq d'(S) \Rightarrow d'(T) \). □

For the next part of the discussion we assume that \( D = \langle R_1 : [A_1^1, \ldots, A_1^{m_1}], \ldots, R_n : [A_n^1, \ldots, A_n^{m_n}] \rangle \) is a fixed flat relational database schema and \( R : [A_1, \ldots, A_m] \) a fixed flat relation schema.

One more technical observation is needed.

**Lemma 4.11.** If \( Q \) is a BK-program from \( D \) to \( R \) which is defined on all inputs not involving null values, then it is defined on all inputs.

**Proof.** Suppose that \( Q' \langle d \rangle \) is defined for all inputs \( d \) over \( D \) with no null values. Suppose that \( d' \) is an input with null values. Choose an atom \( c \notin \text{dom}(d') \) and let \( d \) be the result of replacing \( \bot \) by \( c \) everywhere in \( d' \). Then \( Q' \langle d \rangle \) is also defined. Since \( \mu \) is projecting, Lemma 4.9(2) implies that \( Q' \langle d' \rangle \) is also defined (and \( = \mu(Q' \langle d \rangle) \)). □

We are now ready to prove Theorem 4.4.

**Proof of Theorem 4.4.** This argument uses a technique reminiscent of tableaux arguments for conjunctive queries. Let \( a_1, \ldots, a_m \) be \( m (= \text{the arity of } R) \) distinct constants, \( z_1, \ldots, z_m \) distinct variables, and \( \lambda \) the mapping with domain \( \{ a_1, \ldots, a_m \} \) such that \( \lambda(a_j) = z_j \) for \( 1 \leq j \leq m \). A template is an instance \( T \) of \( D \) (possibly with null
values) such that \( \text{adom}(T) \subseteq \{a_1, \ldots, a_m\} \). For each template \( T \), Lemma 4.11 implies that \( Q^*(T) \) is defined. Let \( Q[T] = \{t_1^T, \ldots, t_{pt}^T\} \). For each \( 1 \leq i \leq pt \), let \( r_i^T \) be the CQ-rule (abusing the notation in the natural manner):

\[
[R : \{\lambda(t_i^T)\}] \leftarrow \lambda(T).
\]

The program \( Q^* \) is defined to be the set of all such CQ-rules:

\[
Q^* = \{r_i^T \mid T \text{ a template and } 1 \leq i \leq pt\}.
\]

**Claim 4.12.** For each input instance \( d \) (possibly with null values), each template \( T \) and each \( 1 < i < pt \), \( r_i^T(d) \leq Q[d] \).

To demonstrate this claim, let \( d, T, \) and \( i \) be fixed, and let \( r_i^T = \phi \leftarrow \phi' \). Suppose that \( s \in \{\sigma(\phi) \mid \sigma(\phi') \subseteq d\} \). It suffices to show that \( s \leq t \) for some \( t \in Q[d] \). Let \( \sigma \) be chosen so that \( s = \sigma(\phi) \) and \( \sigma(\phi') \subseteq d \). By the definition of \( r_i^T \), this implies that \( \sigma \cdot \lambda(T) \leq d \). By Lemma 4.9 (a) and the monotonicity of BK program application we have:

\[
\sigma \cdot \lambda(Q^*(T)) \subseteq Q^*(\sigma \cdot \lambda(T)) \subseteq Q^*(d).
\]

Thus \( \sigma \cdot \lambda(Q[T]) \equiv \sigma \cdot \lambda(Q[T]) \subseteq Q[d] \).

Now, \( s = \sigma(\phi) = \sigma \cdot \lambda(t_i^T) \), and \( t_i^T \in Q[T] \). This implies that there is some \( t \) such that \( s = \sigma \cdot \lambda(t_i^T) \leq t \in \sigma \cdot \lambda(Q[T]) \subseteq Q[d] \). It follows that \( r_i^T(d) \leq Q[d] \), and so Claim 4.12 is established.

Because \( Q^* \) is finite, it easily follows from the above claim that \( Q'[d] \subseteq Q[d] \) for each input instance \( d \). For the opposite inclusion, we establish the following claim.

**Claim 4.13.** For each input instance \( d \), \( Q[d] \subseteq Q'[d] \).

Let \( d \) be fixed, and suppose that \( t = [A_{i,1}; b_1, \ldots, A_{i,l}; b_l] \in Q[d](R) \) (where the \( b_i \)'s are not necessarily distinct). Let the homomorphism \( \mu \) be defined so that \( \mu(b_i) = b_i \) for each \( 1 \leq i \leq l \), and \( \mu(c) = \bot \) for all other constants \( c \). Note that \( \mu \) is projecting. Letting \( d' = \mu(d) \), this implies that \( d' \leq d \) (Lemma 4.7(4)) and \( \mu(Q[d]) = Q[\mu(d')] = Q[d'] \) (Lemma 4.9(2)). Since \( t \in Q[d] \) and \( \mu(t) = t \), there is some \( t' \) such that

\[
t \leq t' \in \mu(Q[d]) = Q[d'].
\]

Since \( \text{adom}(Q[d']) \subseteq \text{adom}(d') \subseteq \{b_1, \ldots, b_l\} \), we have \( t = t' \), and so \( t \in Q[d'] \).

Since \( d' \) is an instance over \( \{b_1, \ldots, b_l\} \) and \( l \leq m = \text{the arity of } R \), there is some template \( T \) and some isomorphism \( \xi \) mapping \( d' \) to \( T \). Since \( t \in Q[d'], \xi(t) \in Q[T] \). Let \( i \) be chosen so that \( \xi(t) = t_i^T \). Then \( \xi(t) \in r_i^T(T) \). Lemma 4.8(1) now implies that \( t = \xi^{-1} \cdot t \in \xi^{-1}(r_i^T(T)) \leq r_i^T(\xi^{-1}(T)) = r_i^T(d') \). Thus, \( t \in Q'[d'] \). Since \( d' \leq d \), the monotonicity of BK programs implies that there is some \( t'' \) such that \( t \leq t'' \in Q[d] \). Since \( Q^*(d') \subseteq Q^*(d) \) and \( t \in Q[d] \), this implies that \( t = t'' \) and so \( t \in Q'[d] \) as desired. This completes the proof of Claim 4.13, and of the theorem.

The characterization given in Theorem 4.4 focuses exclusively on BK-programs which are known to be defined on all of their inputs. In the following we explore the issue of BK-programs with flat input and output which are undefined on some inputs, and then present an extension of the above theorem which incorporates some of these cases. First, we observe
Proposition 4.14. It is undecidable, for a BK-program \( Q \) from \( D \) to \( R \) whether \( Q[I_0] \) is defined, where \( I_0 \) denotes the input instance such that \( Z_0(R_i) = 0 \) for each \( 1 \leq i \leq k \).

Proof. This result relies on the proof of Theorem 4.21, which is presented below and which does not rely on this result. It is shown in that proof how BK programs can be constructed to simulate arbitrary (conventional) Turing machines. We use that construction here. In particular, for each Turing machine \( M \) let \( Q_M \) be a BK-program which ignores the input, and which simulates the operation of \( M \) on the empty string. In particular, if \( M \) halts, let \( Q_M \) give as output \( R: \emptyset \). Then \( Q_M[I_0] \) is defined iff \( M \) halts on the empty string; this reduction demonstrates the proposition. \( \square \)

The following example illustrates two ways that a BK-program \( Q \) might be defined on some inputs and undefined on others.

Example 4.15. (a) Let \( Q_1 \) be a BK-program from \( D \) to \( R \), and let \( i_1, \ldots, i_l \) be a subsequence of \( 1, \ldots, n \). Suppose that \( Q \) includes the rule

\[
[Sim_M : \{ \}] \leftarrow [R_{i_1} : \{x_1\}, \ldots, R_{i_l} : \{x_l\}]
\]

and also rules that simulate the operation of a Turing machine \( M \) on the empty string when the condition \([Sim_M : \{x\}]\) is met. Assuming that the only part of \( Q \) which has the potential for leading to an undefined answer are the rules for simulating \( M \), \( Q \) will be undefined on input \( d \) iff: (i) \( I(R_{i_j}) = \emptyset \) for each \( 1 \leq j \leq l \), and (ii) \( M \) does not halt on the empty string. It follows that it is undecidable whether a BK-program will be defined on inputs for which a given subset of the input relations are non-empty.

(b) Let \( Q_2 \) be a BK-program from \( R_1 \) to \( R \) be defined by the rules:

\[
[B : x] \leftarrow [R_1 : \{[A : x]\}],
[R : \{[A : x]\}] \leftarrow [B : x].
\]

Then for each input \( d \) over \( R_1 \),

\[
Q[d](K) = \begin{cases} 
\pi_A d(R_1) & \text{if } |\pi_A d(R_1)| \leq 1, \\
\text{undefined} & \text{otherwise}.
\end{cases}
\]

This is easily verified for the case where \( |\pi_A d(R_1)| \leq 1 \). Suppose that \( |\pi_A d(R_1)| \geq 2 \), and that \([A : a]\) and \([A : b]\) are \( \leq d(R_1) \). The result of the first application of the rules of \( Q \) is

\[
\bigcup \{[R_1 : d(R_1), B : a], [R_1 : d(R_1), B : b]\} = \top
\]

and so subsequent application of \( Q \) yields undefined. As in part (a), this behavior can be used in conjunction with Turing machine simulation to make the issue of \( Q \) being defined undecidable.

The kind of situation arising in Example 4.15(b) is not possible in BK-programs satisfying the following syntactic condition.

Definition 4.16. A BK-program \( Q \) from \( D \) to \( R \) is syntactically \( \top \)-less if each rule \( r \) in \( Q \) has the form
Lemma 4.17. If a BK-program $Q$ from $D$ to $R$ is syntactically $\top$-less, then for each input $d$, $Q^*(d)$ is defined iff $Q^d(d)$ is defined, where $\hat{d}$ is defined by

$$\hat{d}(R_i) = \begin{cases} \{d\} & \text{if } d(R_i) \neq \emptyset \\ \emptyset & \text{otherwise.} \end{cases}$$

Proof. Since $\hat{d} \leq d$, if $Q^*(\hat{d})$ is undefined so is $Q^*(d)$. Suppose now that $Q^*(d)$ is undefined. A straightforward induction based on the structure of the rules in $Q$ shows for each $i \geq 0$ that $T_Q \uparrow_i(d)$ is defined and $\neq \top$. Thus, the $T_Q \uparrow_i(d)$ sequence is infinite and monotonically increasing.

Since $\text{adom}(T_Q \uparrow_i(d)) \subseteq \text{adom}(d) \cup \text{adom}(Q)$ for each $i \geq 0$ and the $Q$ is syntactically $\top$-less, there is a monotonically increasing subsequence $i_1, i_2, \ldots$ such that $\text{depth}(T_Q \uparrow_i(d)) > \text{depth}(T_Q \uparrow_{i+1}(d))$ for each $j \geq 1$. Let the homomorphism $\mu_i$ be defined so that $\mu_i(a) = \bot$ for each atom $a$. Note that $\mu_i$ is projecting and $\mu_i(d) = d$. Lemma 4.8(b) now implies that $T_Q \uparrow_i(d)$ does not converge, and $Q^*(d)$ is not defined. $\square$

Definition 4.18. A looping rule from $D$ to $R$ has one of the following forms (where $S \neq R$ is a fixed attribute name not in $D$):

$$[S : \{[H: x]\}] \leftarrow [R_i : \{x_i\}],$$

$$[S : \{[H: a, T: x]\}] \leftarrow [S : \{x\}],$$

where $i_1, \ldots, i_l$ is a subsequence of $1, \ldots, n$ and $a \in U$.

It is now straightforward to demonstrate the following generalization of Theorem 4.4 to syntactically $\top$-less BK-programs.

Corollary 4.19. If $Q$ from $D$ to $R$ is a syntactically $\top$-less BK-program, then there is a BK-program $Q'$ consisting of $CQ$-rules and looping rules from $D$ to $R$ such that for each input $d$ (possibly with null values), either both $Q'[d]$ and $Q'[d]$ are undefined, or both are defined and $Q[d] = Q'[d]$.

We now turn to the second main result of the section, and present a construction showing the BK-programs can simulate Turing machines, if given a suitably encoded input. We first introduce some notation, and illustrate a mechanism for simulating the testing of equality of constants within the BK-calculus.

We assume that $F \in A$. Also, for each $g \in A$, let $\check{g}$ denote the object $[g : [\ ]] = [g : [F : \bot]]$. If $v$ is a variable, then $\check{g}(v)$ denotes the term $[g : [F : v]]$.

Example 4.20. Let $f, g, h \in A$ be distinct and $r$ the following rule:

$$[OUT : \{[A : \check{h}(v)]\}] \leftarrow [IN : \{[A : \check{f}(v)]\}].$$
Then
\[ r([IN : \{A: \hat{f}\}]) = [OUT : \{A: \hat{f}(v)\}] , \]
\[ r([IN : \{A: \hat{g}\}]) = \bot \, . \]
In the first case, a substitution \( \sigma \) satisfies \( \sigma([IN : \{A: \hat{f}(v)\}]) < [IN : \{A: \hat{f}\}] \) only if \( \sigma(v) = \bot \). In the second case, there are no variable assignments \( \sigma \) such that \( \sigma([IN : \{A: \hat{f}(v)\}]) < [IN : \{A: \hat{g}\}] \). This illustrates how objects \( \hat{g} \) for \( g \in A \) can be used as constants.

**Theorem 4.21.** The language of BK is complete in the set of Turing computable functions, i.e., for each (conventional) Turing machine \( M \) there is a BK-program \( Q_M \) that simulates \( M \). Furthermore, \( Q_M \) can be constructed in polynomial time in the size of \( M \).

To prove the theorem we describe a framework for simulating Turing machines with BK. The proof is concluded after demonstrating a key lemma (Lemma 4.22).

We now describe how BK-program can be set up to simulate Turing machines. To this end, suppose that \( M = (K, \Sigma, \Gamma, \delta, s_0, s_h) \) is a (conventional) deterministic Turing machine with states \( K \), input/output alphabet \( \Sigma \), tape alphabet \( \Gamma \supset \Sigma \), start state \( s_0 \) and halting state \( s_h \), and that \( M \) computes a (partial) function from \( \Sigma^* \) to \( \Sigma^* \). In particular, we assume that \( M \) is started in state \( s_0 \) reading the blank symbol immediately left of the input, and that if it halts then the tape is blank except for the output, and the tape head is reading the blank symbol immediately left of the output string. (The assumptions about input and output IDs are for technical convenience only.)

Let \( Con = K \cup \Gamma \cup \{\$\} \), where \( \$ \notin \Gamma \), and suppose without loss of generality that \( Con \subseteq A \). For each string \( \alpha = a_1a_2 \ldots a_n \in \Gamma^*, n > 0 \), let \( \hat{\alpha} \) denote the object
\[ [H: \hat{a}_1, [H: \hat{a}_2, \ldots [H: \hat{a}_n, T: [H: \hat{\$}] \ldots ]]] \]

For each integer \( i > 0 \), let \( i \) denote \( i \). Note that \( i \neq j \) and \( j \neq i \) whenever \( i \neq j \).

We now describe a BK-program \( Q_M \) that simulates \( M \). The main component in \( Q_M \) is the 5-ary relation \( ID \), which will hold IDs of \( M \) along with a counter indicating when the ID was reached. On input \( \alpha \in \Sigma^* \) for \( M \) we use as input for \( Q_M \) the object
\[ o_\alpha = \{\{ State : s_0, \{ Left : [H: \$], Letter : \hat{b}, Right : \hat{z}, Ctr : [H: \$] \}\} \} \]

Intuitively, after applying \( Q_M \) there will be a correspondence between 5-tuples in \( ID \) and \( ID \)'s that \( M \) reaches when started on input \( \alpha \). In particular, the 5-tuple
\[\text{State: } \tilde{s}, \text{ Left: } \tilde{b}, \text{ Letter: } \tilde{a}, \text{ Right: } \tilde{r}, \text{ Ctr: } \tilde{i}\] corresponds to the ID \((s, \beta^g, a, \gamma)\) reached on the \(i\)th step of the computation of \(M\) on \(x\).

In \(Q_M\) we include the rule

\[
\text{OUTPUT: } y \leftarrow \text{ID: } \left\{ \begin{array}{l}
\text{State: } \tilde{q}(v), \\
\text{Left: } [H: \tilde{s}(v)], \\
\text{Letter: } \tilde{b}(v), \\
\text{Right: } y, \\
\text{Ctr: } [H: \tilde{s}, T: z]
\end{array} \right\}
\]

As we shall see, if \(M\) halts on the input \(x\), then \(Q'_M(x)\) is defined and \text{OUTPUT} holds the encoded output of \(M\).

\(Q_M\) will include one or more rules for each value of the transition function \(\delta\). Suppose that \(\delta(q, a) = (q', b, -)\). Then we include the rule

\[
\text{ID: } \left\{ \begin{array}{l}
\text{State: } \tilde{q}(v), \\
\text{Left: } x, \\
\text{Letter: } \tilde{b}(v), \\
\text{Right: } y, \\
\text{Ctr: } [H: \tilde{s}, T: z]
\end{array} \right\} \rightarrow \text{ID: } \left\{ \begin{array}{l}
\text{State: } \tilde{q}(v), \\
\text{Left: } x, \\
\text{Letter: } \tilde{a}(v), \\
\text{Right: } y, \\
\text{Ctr: } z
\end{array} \right\}
\]

Suppose now that \(\delta(q, a) = (q', b, L)\). Then for each \(c \in \Gamma\) we include the rule

\[
\text{ID: } \left\{ \begin{array}{l}
\text{State: } \tilde{q}(v), \\
\text{Left: } x, \\
\text{Letter: } \tilde{b}(v), \\
\text{Right: } [H: \tilde{b}(v), T: y], \\
\text{Ctr: } [H: \tilde{s}, T: z]
\end{array} \right\} \rightarrow \text{ID: } \left\{ \begin{array}{l}
\text{State: } \tilde{q}(v), \\
\text{Left: } [H: \tilde{c}(v), T: x], \\
\text{Letter: } \tilde{a}(v), \\
\text{Right: } y, \\
\text{Ctr: } z
\end{array} \right\}
\]

and also the rule

\[
\text{ID: } \left\{ \begin{array}{l}
\text{State: } \tilde{q}(v), \\
\text{Left: } [H: \tilde{s}(v)], \\
\text{Letter: } \tilde{b}(v), \\
\text{Right: } [H: \tilde{b}(v), T: y], \\
\text{Ctr: } [H: \tilde{s}, T: z]
\end{array} \right\} \rightarrow \text{ID: } \left\{ \begin{array}{l}
\text{State: } \tilde{q}(v), \\
\text{Left: } [H: \tilde{s}(v)], \\
\text{Letter: } \tilde{a}(v), \\
\text{Right: } z, \\
\text{Ctr: } z
\end{array} \right\}
\]

Analogous rules are included for transitions in which the tape head moves right.

**Lemma 4.22.** For each \(i \geq 0\),

\[
\text{State: } \tilde{q}, \text{ Left: } \tilde{b}, \text{ Letter: } u', \text{ Right: } \gamma', \text{ Ctr: } i
\]

iff there is a computation with length \(i\) of \(M\) on input \(x\) that ends with the \(\text{ID}(q, \beta, u, \gamma)\), where \(q' = \tilde{q}, \beta' = \tilde{\beta}, u' = \tilde{u}, \gamma' = \tilde{\gamma}, \) and \(i = \tilde{i}\).
Proof. Let \( o_j \) denote \( T_Q \uparrow_j (o_2) \) for each \( j \geq 0 \). The claim is clearly true for the case \( j = 0 \). Suppose inductively that the claim holds for \( i - 1 \) where \( i > 0 \) is fixed. For the if direction, suppose that there is a computation of \( M \) on input \( \alpha \) with length \( i \) which ends with the ID \( (q, \beta, u, \gamma) \), and define \( q', \beta', u', \gamma' \) from \( q, \beta, u, \gamma \) as in the claim. Let \( (q_1, \beta_1, u_1, \gamma_1) \) be the ID occurring in the computation immediately before \( (q, \beta, u, \gamma) \). Letting \( q_1', \beta_1', u_1', \gamma_1' \) be defined from \( q_1, \beta_1, u_1, \gamma_1 \) in the natural fashion, the inductive assumption implies that \( \text{State} : q_1', \text{Left} : \beta', \text{Letter} : u', \text{Right} : \gamma' \in o_{i-1}(ID) - o_{i-2}(ID) \) (Note that \( o_j(ID) = \emptyset \) for \( j < 0 \) by definition). Using the value of \( \delta \) which takes \( (q_1, \beta_1, u_1, \gamma_1) \) to \( (q, \beta, u, \gamma) \) it is now easy to find the rule of \( Q_M \) which justifies the inclusion of \( \text{State} : q', \text{Left} : \beta', \text{Letter} : u', \text{Right} : \gamma', \text{Ctr} : i \) into \( o_i(ID) \).

Now suppose that \( t_1 = \text{State} : q_1', \text{Left} : \beta', \text{Letter} : u', \text{Right} : \gamma', \text{Ctr} : i \) which ends with the ID \( (q, \beta, u, \gamma) \) in \( o_i(ID) - o_{i-1}(ID) \). Since \( t_1 \) does not involve the set construct (and by the form of the rules in \( Q_M \), \([ID: \{t\}] \in \{\sigma(\phi) \mid \phi \leftarrow \phi' \} \) is a rule in \( Q_M \) and \( \sigma(\phi') \leq o_{i-1} \) ).

By the form of the rules in \( Q_M, \sigma(\phi') = [ID: \{s_1\}] \) for some \( s_1 = \text{State} : q_1', \text{Left} : \beta', \text{Letter} : u', \text{Right} : \gamma', \text{Ctr} : t_1' \). Because \( \sigma(\phi') \leq o_{i-1} \) there is some \( s_2 = \text{State} : q_2', \text{Left} : \beta', \text{Letter} : u', \text{Right} : \gamma', \text{Ctr} : t_2' \) such that \( q_2 = q_2', \beta_2 = \beta', u_2 = u', \gamma_2 = \gamma', \) and \( t_2 = i - 1 \); and a computation of \( M \) on \( z \) with \( i \) steps ending with the ID \( (q_2, \beta_2, u_2, \gamma_2) \). Given the form of \( \phi' \) and since \( \sigma(\phi') = s_1 \leq s_2 \), it is easily seen that there is an assignment \( \sigma_2 \) such that \( \sigma_2(\phi') = s_2 \). Let \( t_2 \) be chosen so that \( \sigma_2(\phi) = [ID: \{t_2\}] \). From the choice of \( \sigma_2 \) we have \( t_1 \leq t_2 \) and \( t_2 \in o_i(ID) \). Since \( t_1 \in o_i(ID) \) this implies that \( t_1 = t_2 \). Finally, the definition of rules in \( Q_M \) and the choice of \( \sigma_2 \) imply that \( i = i \) and that there are \( q, \beta, u, \gamma \) corresponding to \( q', \beta', u', \gamma' \) with the desired properties. \( \Box \)

Proof of Theorem 4.21. Let \( M \) and \( x \) be fixed. Suppose that the computation of \( M \) on \( o_x \) does not terminate. Then the \( T_Q \uparrow_i (o_x) \) sequence is monotonically increasing (in particular, \( \text{depth}(T_Q \uparrow_i (o_x)) \geq \text{depth}(i) + 3 = i + 6 \) for each \( i \geq 0 \)). Thus, \( Q'_M(o_x) \) is undefined.

Suppose on the other hand that \( M \) halts in \( ID(q_h, \in, \beta, \gamma) \) after \( i \geq 0 \) steps. Then \( \text{State} : q_h, \text{Left} : \in, \text{Letter} : \beta', \text{Right} : \gamma, \text{Ctr} : i \) is in \( T_Q \uparrow_i (o_x)(ID) - T_Q \uparrow_{i-1} (o_x)(ID) \). Also, since \( M \) has no move out of \( q_h, T_Q \uparrow_{i+1} (o_x) = T_Q \uparrow_i (o_x) = T_Q \uparrow_0 (o_x) = Q'(o_x) \) is defined. It now follows that \( Q'(o_x)(\text{OUTPUT}) = \gamma \) as desired. \( \Box \)

5. Conclusions

Query languages with recursive types have been studied in [21,34] and in this paper. In particular, this and the companion paper [24] considered a data model that extends the complex object model of [3] with a hereditarily finite set construct and their query languages in the well known three styles: algebra and calculus [34] and deductive (this paper). Two kinds of deductive languages are studied. The query languages of the first kind are natural extensions of COL [5] (or the recursive language of [3]). For these languages, the addition of recursive types raises the expressive power to the class of all computable queries. The results are not surprising and the role the recursive types play is similar to that of invented values [9,15].
The second kind of languages are a result of applying the theory of domains traditionally used in the denotational semantics of programming languages to the database context [12]. For this reason, it is interesting to understand the impact the domain theory on query languages in terms of their expressiveness and complexity. We studied in detail the BK-calculus as a representative in this class and found that it is rather unique. On one hand, it loses the ability to express the natural join; on the other hand, its data complexity goes too arbitrarily high. The latter is seemingly due to the recursive types in a similar way to the COL extensions. The former appears to be a consequence of using the Hoare ordering for sets [12]. It remains an interesting question to fix the problem and we believe [12] may be a good starting point.

Acknowledgements

The authors thank Peter Buneman and Aaron Watters for showing us provocative examples concerning the relationship between natural join and the BK calculus, which ultimately led to the development of Theorem 4.4; and also the anonymous referees for their comments on an earlier version of the paper.

References


