Untyped Sets, Invention, and Computable Queries*

Richard Hull and Jianwen Su

Computer Science Department†
University of Southern California
Los Angeles, CA 90089-0782

Hull@cse.usc.edu JSu@cse.usc.edu

Conventional database query languages are considered in the context of untyped sets. The algebra without **while** has the expressive power of the typed complex object algebra. The algebra plus **while**, and COL with untyped sets (under stratified semantics or inflationary semantics) have the power of the computable queries. The calculus has power beyond the computable queries; and is characterized using the typed complex object calculus with invention. The Bancilhon-Khoshafian calculus is also discussed. A technical tool, called "generic Turing machine", is introduced and used in several of the proofs.

1 Introduction

Since the relational data model [Cod70] has its significant drawbacks in many application areas (such as engineering design, CAD, etc.), various attempts have been made towards extending the model and its query languages (calculus and algebra) [HK87]. Among these proposed models and languages, one important common feature is to use the "set" (or "grouping") construct. Roughly speaking, there are two fundamentally different ways to introduce sets: (1) typed sets, which require all elements in a set to have certain fixed structure(s), as in the nested relation and most complex object models [Hul87, AB88, AG87]; and (2) untyped sets, which impose no restrictions on the types of their members, e.g., FAD [BBKV87], the Bancilhon-Khoshafian calculus (BK) [BK86], and also the "Set Theoretic Data Model" of Gemstone [CM84]. In this extended abstract we conduct a theoretical study of expressive power of several query languages of complex objects using untyped sets. Our results indicate that untyped sets generally yield considerably more expressive power, and in several cases, the power of computable queries. Also, several pairs of query languages (e.g., complex object calculus and algebra) which are equivalent (nonequivalent) in the conventional context are seen to be nonequivalent (equivalent) in the context of untyped sets. Finally, we indicate how analogous results can be obtained in the presence of lists or freely interpreted function symbols.

The investigation described here provides a bridge between research on the use of (typed) sets in query languages [HS88b, KV88, PvG88, GvG88] and research on computable queries [CH80, AV87]. While the complex object and nested relation models where developed several years ago, it is only recently that theoretical investigations of the expressive power and complexity of their associated query languages have been made. For example, the independent investigations [HS88b, KV88] show that each level of nested sets gives more expressive power at an exponential cost (of time, space); and [AB88, GvG88] show that in the context of the nested relation algebra, the powerset construct is equivalent to various iterative constructs (e.g., fix-

*This work supported in part by NSF grant IRI-87-19875 and DARPA contract MDA903-81-C-0335. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official opinion or policy of NSF, DARPA, the U.S. Government, or any other person or agency connected with them.

†Part of this work was performed while the authors were visiting the Information Sciences Institute of USC.

point, **while**). On the other side, [AV87] introduced a transaction language which has the expressive power of (essentially) the computable queries originally introduced in [CH80]. In the current paper, we show that if untyped sets are permitted, then the algebra (with **while**) and deductive languages (with negation, using either stratified or inflationary semantics) also have the power of the computable queries. Furthermore, the calculus has expressive power beyond this class.

Calculus, algebraic, and deductive languages are considered here. In all cases, we restrict attention to their behavior on flat relational input and output. When sets are typed, these languages all have the same expressive power, that is, the expressive power of the class $\mathcal{E}$ of *elementary queries* [HS88a], which are generic mappings from flat relations to flat relations and computable by Turing machines within hyper-exponential time (space).

When incorporating untyped sets it is shown that the algebra (with the powerset operator) is expressively equivalent to the one with typed sets. We also study the algebra extended by a **while** looping construct; in the presence of untyped sets this is closely related to FAD. We show that every computable query is expressible in this extended algebra regardless of the presence of the powerset operator. Thus, untyped sets break the "balance" between powerset and iteration (e.g., **while**).

A second focus concerns deductive languages. Several such languages have been suggested: COL [AG87], BK, LDL1 [BNR*87], complex object DATALOG [AB88], etc. Here we mainly study the expressive power of two extensions of COL with untyped sets under the semantics of stratification and inflation (respectively). We show that inflationary COL with untyped sets, stratified COL with untyped sets, and the class of computable queries are all equivalent. (Analogous results hold for complex object DATALOG.) This provides an interesting contrast to the fact that stratified DATALOG$^\neg$ is strictly weaker than inflationary DATALOG$^\neg$ [Kol87, KP88, AV88]. We also briefly discuss BK. Although we do not characterize its expressive power, we present results indicating some of the limitations of its expressive power.

Finally, we turn to untyped sets in the calculus. It turns out that the untyped sets can be used as invented values [AV87, HS88b]. Indeed, the calculus with untyped sets has the same expressive power as the calculus with countable invention. Unfortunately, their power is far beyond Turing computability and thus unreasonable. A new semantics called "terminal invention" is proposed. It is shown that the class of queries expressible by the calculus with terminal invention is exactly the class of all computable queries.

In this paper we introduce a technical vehicle, called "generic Turing machine" (GTM), which is used in several of the proofs that languages have the power of the computable queries. Unlike conventional Turing machines, GTMs use an infinite input alphabet which corresponds to the underlying domain of (all) database instances considered. The transistion function of GTMs is finitely expressible, and ensures that the computation and its output is "generic". (However, as with the use of conventional Turing machines to describe query functions, the focus is on GTMs whose output is independent of the order of the input.)

Section 2 reviews basic concepts. GTMs are introduced in Section 3. Section 4 introduces the notion of untyped sets, and analyzes the algebra. Deductive languages and the calculus are studied in Sections 5 and 6. This is only an extended abstract. It is assumed that readers are familiar with COL and BK. Proofs are therefore either omitted or briefly sketched.

2 Preliminaries

In this section, we establish terminology for previously studied concepts including types, objects, databases, query functions; several query languages (algebra, deductive, and calculus); and the notion of equivalent expressive power. Notably, all languages considered have the ability to return the "undefined" value (?) as output.

We assume that **U** is a countably infinite *universal domain* of *atomic objects*, **P** is a countably infinite set of abstract *predicate names*, and **U** and **P** are disjoint. Types are defined recursively from the symbol 'U' and the tuple and set constructs:

Definition: The set of *types* is a family of expressions defined recursively by:

(a) the symbol U is the *basic* type;

(b) if T is a type then $\{T\}$ is a *set* type; and

(c) if $T_1, ..., T_n, n \geq 1$ are types then $[T_1, ..., T_n]$ is a *tuple* type.

A type is *flat* if no set construct occurs in it.

The *domain* of a type T, denoted $dom(T)$, is defined in the usual fashion. Each element of $dom(T)$ is an *object* of type T; any finite subset of $dom(T)$ is an *instance* of type T; and $inst(T)$ denotes the family of instances of T.

Definition: A *database schema* is a sequence $D =< P_1 : T_1, \ldots, P_n : T_n >$ where

(a) $P_i \in \mathbf{P}$ for $i \in [1..n]$;

(b) T_i is a type for $i \in [1..n]$; and

(c) $P_i \neq P_j$ if $i \neq j$.

D is *flat* if T_i is flat for $i \in [1..n]$. A (*database*) *instance* of D is a sequence $d =< P_1 : I_1, \ldots, P_n : I_n >$ where I_j is an instance of T_j for each $j \in [1..n]$. The family of instances of D is denoted $inst(D)$.

The *atomic* (*active*) *domain* of an object X (instance I, database instance d), denoted $adom(X)$ ($adom(I), adom(d)$), is the set of atomic objects "used" in building X (I, d).

Definition: If D is a database schema and T is a type, a *query function* f from D to T, denoted $f : D \rightarrow T$, is a (partial) mapping from $inst(D)$ to $inst(T)$. Further, let $d \in inst(D)$. If $f(d)$ is undefined, we write $f(d) =$ '?'. Also, we define $indom(f, d) = adom(d)$, and $outdom(f, d) = adom(f(d))$ ($\emptyset$ if $f(d) =$ '?').

Definition: If $C \subseteq \mathbf{U}$ is a finite set, $f : D \rightarrow T$ is a query function, then: f is

(a) (*input*) *domain preserving wrt* C, if $\forall d \in inst(D)$, $outdom(f, d) \subseteq indom(f, d) \cup C$;

(b) *C-generic* if $f \circ \sigma = \sigma \circ f$ for each permutation[1] σ over $\mathbf{U}$ with $\forall x \in C$, $\sigma(x) = x$.

f is *domain preserving* (*generic*) if f is domain preserving wrt C (C-generic) for some C.

It is easily verified that each generic (and deterministic) query function is domain preserving. All queries in the languages discussed here are generic and domain preserving.

We now introduce two interesting classes of queries. The family of "computable queries" was introduced in [CH80] and also studied in [AV87].

[1] σ is extended naturally to databases.

Definition: The class $\mathcal{C}$ of *computable query functions* is the set of query functions f such that

(a) $f : D \rightarrow T$ is a query function for some flat database schema D and flat type T;

(b) f is generic; and

(c) f is Turing-computable.

We briefly mention a particular framework by which Turing machines are used to compute database mappings. Let D be a flat schema and T a flat type, and $f : D \rightarrow T$ a C-generic database mapping where $C \subset \mathbf{U}$ is finite. A Turing machine M *computes* f if the following conditions are satisfied. First, it is assumed without loss of generality that C and distinquished symbols '0','1',',','(',')','[', and ']' are included in the tape alphabet of M. An input instance I is placed into an ordered list, where values in $adom(I) - C$ are encoded using strings over {0,1}. If M halts, it must hold on its tape the encoding of an instance J of T, such that $adom(J) \subseteq adom(I) \cup C$. Finally, the operation of M must be independent of the encoding for $adom(I) - C$ used, and also independent of the order in which the input is presented.

On the other hand, the notion of "elementary queries" arises naturally in the query languages for complex objects (see Theorem 2.2). Roughly speaking, this family contains queries whose time (space) complexity are elementary functions.

Definition: Let f be a query function and t: $\mathbf{N} \rightarrow \mathbf{N}$ be a function. f has time (space) complexity t, if there exists a Turing machine M which decides $o \in f(d)$ within time (space)[2] $t(\|d\| + \|o\|)$ whenever d is a input database instance and o is an object of the target type.

Notation: The class of *hyper-exponential* functions hyp_i ($i \in \mathbf{N}$) is defined such that $hyp_0(n) = n$ and $hyp_{i+1}(n) = 2^{hyp_i(n)}$, for $i \geq 0$.

Definition: The class $\mathcal{E}$ of elementary query functions is the set of query functions f such that:

(a) $f \in \mathcal{C}$; and

(b) f has hyper-exponential time (space) complexity.

A query is syntactically an expression. With associated semantics, each query expression *realizes*

[2] $\|d\|$ denotes the length of d.

a query function. As our focus is on query functions from flat schemas to flat types, we consider only query expressions of that property in the rest of the abstract.

The algebra used here is essentially equivalent to those of [AB88, KV84, RKS85], but includes a unary operator *undefine* which returns '?' if the instance is empty and the instance otherwise. We use a syntax which views algebra expressions as sequences of assignments, each of which applies a single operator (e.g., in the spirit of [KV84]). This permits the easy incorporation of the **while** construct into the algebra (in the spirit of [GvG88]). An *algebraic query expression* is a sequence of assignments followed by an assignment to a special variable ANS which holds the answer to the query, such that each variable is assigned a value before it is referenced. Let ^{ts}ALG ('ts' for *t*yped *s*ets) denote the family of all such query expressions. The **while** construct has the following format: z := **while** $< x; y >$ **do** assignments **end**, where x, y, z are variables such that z does not occur in the assignments, with the semantics: while the value of y is not empty execute the assignments; z finally gets the value of x at the end of the loop. We consider algebras with both nested **while** and unnested **while**, denoted as ^{ts}ALG+**while** (^{ts}ALG+unnested-**while**). In the evaluation of an algebraic expression, if any variable (or ANS) is assigned the undefined value, or if a while loop does not terminate, then the query evaluates to the undefined value.

In the same spirit as the algebra extending the relational algebra, COL is an extension of DATALOG to incorporate complex objects. Notably, COL uses function symbols which are interpreted as set valued data functions. It has a stratification semantics to ensure a minimal model (see [AG87] for details).

The complex object model in [BK86] is a model based on untyped sets, where 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"[3] the database and then take the least upper bound of the heads.

In our investigation, we view each *query expression* in COL and BK as a query with a special predicate ANS for output. And we use ^{ts}COL, BK to denote the families of query expressions in COL, BK.

In the calculus, formulas are built from $u \approx v$, $u \in v$, $P(u)$ (P is a predicate name, u and v are variables/constants) using sentential connectives ($\wedge$, $\vee$, $\neg$) and typed quantifications ($\forall x/T\phi$, $\exists x/T\phi$) (see [HS88b]). A *calculus query expression* is an expression of form $\{t/T \mid \phi\}$ where ϕ is a (well-typed) formula. Let ^{ts}CALC represent the family of all calculus query expressions. Three different semantics discussed in section 5 are: no invention (limited interpretation), countable invention (unlimited interpretation) [HS88b], and terminal invention.

Definition: 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. If S is a family of query functions, L_1 is *S-equivalent* if

(a) every query expression realizes some f in S, and

(b) every query function in S is realizable.

Thus, the notion of $\mathcal{C}$-equivalent has been called "computationally complete" in other investigations. When the context is clear, both query functions and query expressions are refered as simply *queries*.

The expressive power of many complex object languages is characterized by following two theorems.

Theorem 2.1: [AB88] ^{ts}ALG, ^{ts}COL, ^{ts}CALC, and complex object DATALOG are all equivalent.

Theorem 2.2: ^{ts}CALC is $\mathcal{E}$-equivalent. Hence, ^{ts}ALG, ^{ts}COL, and complex object DATALOG are also $\mathcal{E}$-equivalent.

Proof: (sketch) It is sufficient to show that ^{ts}CALC is $\mathcal{E}$-equivalent. The rest follows easily from this and Theorem 2.1.

To show that every query function realizable in ^{ts}CALC is in $\mathcal{E}$, we use a naive query evaluation algorithm which is easily seen to have hyper-exponential data complexity. For the other direction, let f be an arbitrary query function in $\mathcal{E}$. By definition, there is a Turing machine M which computes f within hyper-exponential time, i.e., hyp_i

[3] In BK, this is based on the sub-object relationship instead of equality.

time for some i. It should be noted that a (partial) computation of M can be represented by a four dimensional array, where the first two columns specify a particular tape square (over time), the third is the symbol on that square, and the last indicates the position of the tape head. In order to "hold" a halting computation of M, the array must be sufficiently large. More precisely, the first two columns must have $hyp_i(n)$ number of distinct elements where n is (roughly) the number of atomic elements in the input. Recall that each level of nested sets induces one high level of exponentiation of data complexity [HS88b, KV88]. Therefore, using the type $\{[T, T, U, U]\}$ where T is a type with i levels of nested sets, the array is able to "hold" all halting computations of M. It is now easy to construct an expression in the calculus to simulate the behavior of M. (More details are provided in [HS88a].) □

3 Generic Turing Machines

As indicated in the introduction, several of the major theorems of the paper show that certain query languages are C-equivalent. In the previous literature such proofs are generally accomplished by showing that a given query langauge can simulate an arbitrary Turing machine which computes a generic query function. In this section we present a technical construct, called "generic Turing machine" (GTM), which can be used to simplify proofs of this sort.

Consider the problem of showing that a given database query language can simulate arbitrary Turing machines which compute a generic query function. Three fundamental issues arise:

1. the tape alphabet of the Turing machine is finite, but the underlying domain of input instances is infinite.
2. the Turing machine must be restricted so that it computes a generic mapping.
3. the computation of the Turing machine must be independent of the order in which the input instance is presented to it.

The notion of generic Turing machine focuses on the first two issues; in particular, a GTM can directly manipulate infinitely many tape symbols, and will necessarily compute only generic mappings. However, we must restrict our attention to GTMs whose output is independent of the order of the input.

Intuitively, a GTM M is a two-tape Turing machine whose alphabet includes the underlying domain $\mathbf{U}$ and a finite set W of *working* (or *punctuation*) symbols. Also, a finite set $C \subset \mathbf{U}$ of constants is explicitly specified in M – these correspond to the constants used in a query, and the computation of M will be C-generic. The transition function for M explicitly uses the members of $W \cup C$ to refer to tape symbols, and also uses the distinquished variables α and β, which are used to refer to elements of $\mathbf{U} - C$.

Formally, we have

Definition: A *generic Turing machine* (*GTM*) (relative to $\mathbf{U}$) is a six-tuple

$$M = (K, W, C, \delta, s_0, h)$$

where

1. K is a finite set of *states.*
2. W is a finite set of *working symbols*, which includes the distinquished symbols ',', '(', ')', '[', and ']' which are used for encoding input relations and output relations.
3. $C \subset \mathbf{U}$ is a finite set of *constants.*
4. $s_0 \in K$ is the *start state.*
5. $h \in K$ is the unique *halting state.*
6. δ is the *transition function* from $(K - \{h\}) \times (W \cup C \cup \{\alpha\}) \times (W \cup C \cup \{\alpha, \beta\})$ to $K \times (W \cup C \cup \{\alpha, \beta\}) \times (W \cup C \cup \{\alpha, \beta\}) \times \{\mathrm{L}, \mathrm{R}, -\}^2$. In a transition value $\delta(q, a, b) = (q', a', b', D1, D2)$, $b = \beta$ only if $a = \alpha$; $\alpha \in \{a', b'\}$ only if $\alpha \in \{a, b\}$; and $\beta \in \{a', b'\}$ only if $\beta \in \{a, b\}$.

M is viewed as having 2 one-way infinite tapes (extending to the right). A transition value $\delta(q, a, b) = (q', a', b', D1, D2)$ is *generic* if $\alpha \in \{a, b\}$. Intuitively, a generic transition value is used as a template for an infinite set of transition values which are formed by letting α (and β if it occurs) range over (distinct) elements of $\mathbf{U} - C$. Assuming this provision, a *computation* of M is defined in the usual fashion.

The restrictions on δ in the above definition ensure that M is deterministic.

Let D be a flat database schema and T a flat type. A GTM M "computes" a query function from D to T in the following manner. An input instance I is enumerated in some order e and placed left-justified on the first of the two tapes of M. M

computes until it reaches the halting state. If the contents of the first tape hold an ordered listing of an instance of T, that instance is the output of the computation of M on input I with order e. If M does not halt, or if the contents of the first tape is not an ordered listing of an instance of T, then M produces the undefined output on I with order e. M is *input-order independent* from D to T if for each instance I of D, the output of M is the same regardless of the input order used for I.

Proposition 3.1: The family of mappings computed by input-order independent generic Turing machines is $\mathcal{C}$-equivalent.

Proof: (Sketch) First, let $f : D \rightarrow T$ be a computable query function which is C-generic for some finite $C \subset \mathbf{U}$. Becuase f is Turing-computable, we can assume that there is a (conventional) Turing machine M which computes f, using the constants C and encoding other elements of U using strings over {0,1}. A GTM M' which simulates M can be constructed as follows: M' includes 0 and 1 in its working symbols. Given an input instance I on its first tape, M' first develops an encoding of $adom(I) - C$ into strings over {0,1}. (The particular encoding will depend on the order of the input to M'.) M' then transforms its initial input into an input for M; simulates M; and then decodes the output.

For the other direction, suppose that $f : D \rightarrow T$ is computed by an input-order independent GTM M. We briefly indicate how to construct a (conventional) Turing machine M' which simulates M. Without loss of generality we can assume that M' is a 4-tape machine; two tapes will be used to simulate each of the tapes of M. M will take as input a listing of an input instance I encoded using strings over {0,1}. Non-generic transitions of M can be used more-or-less directly by M'. Generic transitions of M must be simulated by M'; this is reminiscent of the simulation of an arbitrary Turing machine by one with a two element tape alphabet. □

It is easily verified that if the notion of GTM were modified to have only one tape, then it would be strictly weaker than $\mathcal{C}$. (This is because a 1-tape GTM is unable to replicate elements of $adom(d) - C$.) On the other hand, each n-tape GTM can be simulated by a 2-tape GTM.

4 Algebra with Untyped Sets

To provide a formal model for discussion, the definition of *untyped sets* is first introduced, along with associated definitions (domain, instance, etc.). The algebra described in the previous section is modified to fit this model. Two results are then given. First, it is shown that the algebra (without **while**) with untyped sets is $\mathcal{E}$-equivalent. On the other hand, when (nested or unnested) **while** is included, the algebra is $\mathcal{C}$-equivalent even without powerset. Hence, in this context, **while** is more powerful than powerset, contrasting to the results in the case of typed sets [GvG88].

Definition: A *relaxed type* (*rtype*) is defined as:

(a) U is the *atomic* rtype;

(b) *Obj* is the *universal* rtype;

(c) $\{T\}$ is a *set* rtype if T is an rtype; and

(d) $[T_1, ..., T_n]$ is a *tuple* rtype if $1 \leq n$ and T_i is an rtype for $i \in [1..n]$.

Let **Obj** be the smallest set such that:

(a) $\mathbf{U} \subseteq \mathbf{Obj}$;

(b) $\{X_1, ..., X_n\} \in \mathbf{Obj}$ if $0 \leq n$ and $X_i \in \mathbf{Obj}$ for $i \in [1..n]$; and

(c) $[X_1, ..., X_n] \in \mathbf{Obj}$ if $1 \leq n$ and $X_i \in \mathbf{Obj}$ for $i \in [1..n]$.

Now, the domain of an rtype is defined similarly to that of a type.

Definition: The *domain* of an rtype T, denoted $dom(T)$, is defined:

(a) $dom(U) = \mathbf{U}$;

(b) $dom(Obj) = \mathbf{Obj}$;

(c) $dom(\{T\}) = \{\{X_1, ..., X_n\} \mid 0 \leq n$ and $X_i \in dom(T)$ for $i \in [1..n]\}$; and

(d) $dom([T_1, ..., T_n]) = \{[X_1, ..., X_n] \mid X_i \in dom(T_i)$ for $i \in [1..n]\}$.

Each element in $dom(T)$ is an object. Any finite subset of $dom(T)$ is an instance of T. $inst(T)$ denotes the family of instances of T.

We now make some intuitive comments about the system of rtypes. Most importantly, this system subsumes essentially all of the objects used in BK and FAD. In BK, special "bottom" and "top" objects are used (and FAD permits "bottom"); rtypes do not provide this. Also, both BK and FAD have a convention of named attributes in tuples, while here we use position to identify the coordinates. In the current investigation we use rtypes to provide a minimal typing framework, in order to retain the overall spirit of the typed languages (algebra, COL, calculus) whose variants are studied here. It should be noted that the family of *types* introduced in Section 2 is a proper subset of the family of *rtypes*. Also, unlike types, it is possible for two distinct rtypes to have overlapping domains.

We associate the set **Obj** with an equality relationship extended from the equality of **U** in a natural manner. The algebraic operators are extended in natural ways to range over instances of rtypes. For example, we permit the formation of unions of instances of different rtypes (which result in instances of type *Obj*). Also, horizontal operators such as selection can operate on instances of *Obj*; these "ignore" elements of the instance which do not have the right shape. We use ALG (ALG+(unnested-)**while**) to represent the algebraic language(s) which map from flat schemas to flat types, possibly using rtypes as intermediate types.

The following results show the power of **while** in this context:

Theorem 4.1: (a) ALG and ^{ts}ALG are $\mathcal{E}$-equivalent;

(b) ALG+unnested-**while**–*powerset* and ALG+**while**–*powerset* are $\mathcal{C}$-equivalent.

Proof: (sketch) (a) Obviously, ALG has the full power of ^{ts}ALG and thus all elementary query functions are realizable in ALG. On the other hand, for any given expression E and database d, the number of new objects created during the process of $E[d]$ is an elementary function of the length of E and the number of atomic elements in d since there are no loops. Therefore, E is easily seen to have hyper-exponential data complexity.

(b) It is clear that the algebra with nested or unnested **while** is a procedural and generic language, and thus contained in $\mathcal{C}$. To establish the other directions, we use Theroem 6.4 and show:

(i) $\mathcal{C} \sqsubseteq$ ALG+**while**;

(ii) ALG+**while** $\sqsubseteq$ ALG+**while**–powerset; and

(iii) ALG+**while**–powerset $\sqsubseteq$ ALG+unnested-**while**–powerset.

Among the above three claims, (ii) can be obtained by using a similar reasoning to the proof of a theorem in [GvG88] which states that the powerset construct is equivalent to the **while** construct in the case of the typed algebra. (iii) can be proved by repeatedly collapsing two consecutively nested **while** loops. Roughly speaking, this can be obtained by using a cross product of two condition variables.

We now describe the proof of (i). Let f be a C-generic computable query function from T to D, and let M be an order-independent GTM which computes f. There are three fundamental issues to be addressed in the construction of a query Q in ALG+**while** for simulating M:

(a) encoding the input instance I into a sequence that can be used as the input for M.

(b) providing an arbitrarily large ordered set of indices which can be used to hold the "current" contents of the two tapes of M.

(c) simulating individual steps of M.

In this sketch we address each issue more or less independently; their synthesis is relatively straightforward.

Consider part (a). In ^{ts}ALG it is easy to build a query Q_1 whose output is an object ORD of type $T = [S, S]$ and $S = \{\cdots\{U\}\cdots\}$ (where the nesting is of height k such that $hyp_k(|adom(d)|) \geq \max\{||d||, |adom(d)|!\}$ for all instances d of D); where ORD holds a total order for[4] $cons_S(adom(d))$. Using ORD, we would like to build a binary relation whose first column holds elements of $cons_S(adom(d))$ and whose second column holds a listing of d. In general this is impossible because the operation of Q is generic. For this reason, we simultaneously build and use a family of listings of d, one for each ordering of the set $adom(d) - C$. To this end, we build a query Q_2 in ^{ts}ALG which maps to the type $\{R\}$, where $R = \{[T, U]\}$; and which maps the input d to an object $PERMS = \{\ ENC_\rho \mid \rho$ is a ordering of $adom(d) - C\ \}$.

[4]For a type T and set $X \subset \mathbf{U}$, the *constructive domain* of T relative to X is $cons_T(X) = \{o \mid o$ has type T and $adom(o) \subseteq X\}$.

We now turn to part (b). The query Q will have a crucial **while** loop which will perform part (c), and will simultaneously create one additional element to be used as an index for the tapes of M. The sequence of elements produced will be $a; \{a\}; \{a, \{a\}\}; \{a, \{a\}, \{a, \{a\}\}\}; \ldots$. If the unary relation P holds an initial segment of this sequence, then the (pseudo-)ALG expression[5] $\pi_2\nu_2\sigma_{1=2}(P \times P)) - P$ will hold the least element outside of P.

We turn now to part (c). We first consider the simplified case in which a specific ordering of the input is given in a binary relation IN, whose first column holds an initial segment of $cons_s(adom(d))$ and second column the listing of the input. In this context, we use two 3-ary relations $T1$ and $T2$ and one unary relation S to record the "current" configuration of the computation of M on IN. The first coordinate of $T1$ will hold values from the set P of indices, the second coordinate will hold the symbol in the corresponding tape square of the first tape of M, and the third coordinate will hold either 'Y', indicating that the tape head is there, or 'N', indicating that it isn't. $T2$ will encode the current contents of the second tape analogously. Finally, the unary relation S will hold a single tuple indicating the current state of M. Using these data structures it is straightforward to set up a **while** loop which simulates a single move of M on each iteration, and which terminates if $S(h)$ holds (i.e., if the computation of M is in a halting configuration). A technical detail here is that in the first iterations of the **while** loop, the input stored in IN must be transfered into $T1$.

We conclude by describing how to generalize the above construction so that it simultaneously works with several orderings of the input rather than just one. In particular, for each ordering ρ of $adom(d) - C$ let IN_ρ be a binary relation holding a listing of d in which tuples are ordered lexicographically according to the ordering ρ. It is easy to build a query Q_3 in ^{ts}ALG which, on input d yields the object $\{ [ENC_\rho, IN_\rho] \mid \rho$ is an ordering for $adom(d) - C\}$. Also, rather than using the types $T1$, $T2$ and S described above, we use 2 4-ary relations $\widehat{T1}$ and $\widehat{T2}$ and a binary relation $\widehat{S}$. In each of these, the last column has type R, and will hold elements of $PERMS$. In particular, for each ordering ρ of $adom(d) - C$, the relations $\pi_{1,2,3}\sigma_{4=ENC_\rho}T1$, $\pi_{1,2,3}\sigma_{4=ENC_\rho}T2$, and $\pi_1\sigma_{2=ENC_\rho}S$, will hold the values of $T1$, $T2$, and S for the input IN_ρ. □

[5] Here ν_2 denotes the operation of nesting on the second column.

We conclude this section by the following remark. The "magic power" of untyped sets is its ability to have arbitrarily large, finite sets built without using "invented values" (relative to $indom(d)$). Together with appropriate control structures (e.g., **while**), it then yields rich expressiveness. Since FAD subsumes ALG+**while**, it also is $\mathcal{C}$-equivalent.

5 Deductive Languages with Untyped Sets

The main interests here are two syntactically different languages, ^{ts}COL and BK. We first extend ^{ts}COL to use rtypes rather than types; and study it using two alternative semantics: *stratification* and *inflation*. It turns out that both extensions are $\mathcal{C}$-equivalent. It is clear that each BK query is computable and monotonic. Although we do not characterize the expressive power of BK, we provide a pair of results which indicate significant limitations in its expressive power, at least in the case where it is restricted to map from and to flat relations.

As introduced in [AG87], ^{ts}COL is a strongly typed language: each variable has an associated type. (In practice, a ^{ts}COL program implicitly assumes an assignment of types to variables.) As detailed in the full paper, it is straightforward to modify the syntax of ^{ts}COL to form COL, whose variables range over rtypes. We associate two alternative semantics to COL programs. The first, *stratification* is the natural generalization of the stratified semantics used in ^{ts}COL. The second, *inflation* is the natural generalization of the "inflationary semantics" introduced for DATALOG$^\neg$ in [KP88]. We denote COL extensions with stratification and inflation semantics as COL^{str} and COL^{inf} respectively. In the context of untyped sets, it is possible using either semantics to write a COL program which, on a given input, does not yield a finite minimal model. In this case, we view the output to be *undefined*. (Technically, there is no least fixed-point in such cases, because infinite objects and instances are not permitted in our model.)

In contrast to Theorem 2.2, we have

Theorem 5.1: COL^{str} and COL^{inf} are $\mathcal{C}$-equivalent.

Proof: (Sketch) It is clear that both variants of COL are within $\mathcal{C}$. We discuss the proof that COL^{str}

subsumes $\mathcal{C}$. This follows by an argument similar to the one showing that $\mathcal{C} \sqsubseteq$ ALG+**while** in the proof of Theorem 4.1 (b). As in that argument, let M be a GTM, and recall that three fundamental issues were addressed:

(a) encoding the input instance I into a sequence that can be used as the input for M.

(b) providing an arbitrarily large ordered set of indices which can be used to hold the "current" contents of the two tapes of M.

(c) simulating individual steps of M.

Because ^{ts}COL can simulate ^{ts}ALG, part (a) can be accomplished. Part (c) will be accomplished in a manner similar to the way it was done in the earlier proof, with one wrinkle. Specifically, in the current context the relations $T1$, $T2$, and S will record the entire history of the computation, rather than simply the "current" configuration. (If only the "current" configuration is stored, than negation is needed to make each move, it appears impossible to construct a stratified program which behaves correctly.) To store the entire history, we add another column to each of the relations, which intuitively holds an index indicating which step of the computation is described by the associated tuple.

For part (b), we assume that a is a distinquished constant, and F a function symbol. The ordered set $a; \{a\}; \{a, \{a\}, \{a, \{a\}\}; \ldots$ is now created in the set $F(a)$ by the following rules:

$$a \in F(a) \quad \leftarrow$$
$$\{a\} \in F(a) \quad \leftarrow$$

and the set of rules

$$\{\{u\} \in F(a) \leftarrow u \in F(a), S(t,p,s) \mid s \text{ is a nonhalting state of } K\}$$

(Here t will typically hold an element of IN_ρ denoting the time; if $S(t,p,s)$ is true at some point in the execution of the program, then M is in the state q during the u-th step in the computation.) □

We now turn to BK, which is obviously monotonic and contained in $\mathcal{C}$. It is natural to compare the expressive power of BK with the class of monotonic query functions in $\mathcal{E}$ and/or $\mathcal{C}$. It should be noticed that BK has two main differences from COL. The first, that negation is not included, has already been mentioned. The second involves variable instantiation: in BK, variable instantiation is not based on equality, but rather on sub-object relationship.

The following discussion indicates the difficulties in using the techniques above to analyze the expressive power of BK.

Example 5.2: Suppose the database d has two binary relations: R_1 of type {[A:U, B:U]}, and R_2 of type {[B:U, C:U]}. (We use here a variant of the BK syntax.) Consider the following rule:

$$R\{[A{:}x, C{:}z]\} \leftarrow$$
$$R_1\{[A{:}x, B{:}y]\},\ R_2\{[B{:}y, C{:}z]\}$$

At first glance, it appears that this rule computes the join of R_1 and R_2. However, by using variable instantiations in which y is assigned $\perp$, it is clear that if R_1 and R_2 are nonempty then R will hold $\pi_1 R_1 \times \pi_2 R_2$. □

Indeed, the following result shows that BK cannot perform joins.

Proposition 5.3: There is no BK query which takes as input two binary relations R_1 of type {[A:U, B:U]} and R_2 of type {[B:U, C:U]} and as output produces the natural join of R_1 and R_2.

Proof: (sketch) Suppose that the BK query Q has the property that $Q[R_1 : I_1, R_2 : I_2] \supseteq I_1 \bowtie I_2$ for all input relations I_1 and I_2. Let $I_1 = \{[A : 1, B : 2]\}$ and $I_2 = \{[B : 2, C : 3], [B : 4, C : 5]\}$. Then $[A : 1, C : 3] \in Q[I_1, I_2]$. Using results from [BK86], there is a tree of rules which "derives" the tuple $[A : 1, C : 2]$ in the output. Transform this tree by replacing all instantiation assignments of variables to 2 by an assignment to $\perp$, and all assignments to 3 by assignments to 5. This will yield a derivation of the tuple $[A : 1, C : 5]$, whence Q does not compute the join. □

In the proof that COL $\equiv \mathcal{C}$ we simulated Turing computations by constructing and using an ordered sequence of objects. The following example explores the problems that arise in attempting to construct sequences in BK.

Example 5.4: Suppose that the relation S with structure {[A:U, B:U]} is used to hold a chain of values, starting with the distinquished symbol '$' and ending with the distinquished symbol '#'. (E.g., the instance {[$,1], [1,2], [2,3], [3,#]} holds a chain

of length 5). We now consider a naive attempt to build a BK program which constructs a list which holds the chain held by S (e.g, in this case yielding the list [H:#, T:[H:3, T:[H:2, T:[H:1, T:$]]]]). Consider the following pair of rules:

$$LIST\{[H:x, T:\$]\} \leftarrow S\{[A:\$, B:x]\}$$
$$LIST\{[H:x, T:[H:y, T:z]]\} \leftarrow$$
$$S\{[A:y, B:x]\}, LIST\{[H:y, T:z]\}$$

The second rule involves a join-like condition on S and $LIST$. It is easily verified that on an input for S which includes at least the tuple [$,1], this pair of rules will produce the infinite sequence of lists: [H:⊥, T:$], [H:⊥, T:[H:⊥, T:$], [H:⊥, T:[H:⊥, T:[H:⊥, T:$], ... In particular, the execution of this program will not terminate, and so its output is undefined. □

Using a generalization of the argument of Proposition 5.3, we also have:

Proposition 5.5: There is no BK query which takes as input a binary relation S holding a chain (starting with $ and ending with #) and produces as output a list (as in Example 5.4) holding that chain.

Proof: If the output can compute $[H:\#, T:[H:n, T:[H:n-1, T:[\ldots H:2, T:[1,\$]]\ldots]]]$ from an input chain I_S, note that the tuples [$,⊥],[⊥,⊥] are also "in" I_S. □

The above results indicate the difficulty of using BK to simulate queries from other languages and also other computational formalisms, such as Turing machines. At present, the problem of characterizing the expressive power or complexity of BK queries remains open.

6 Untyped Sets = Invention for Calculus

In this section, we explore how untyped sets yield more expressive power for calculus. This turns out to be related to the study of (complex object) calculus under unlimited interpretation, which was analyzed using invented values in [HS88b]. In the current paper, the calculus with untyped sets is characterized by a known class of calculus under invention. However, the functions realized by these languages are not computable, in fact, not even recursively enumerable. Thus, a new semantics called "terminal invention" is proposed. And it is shown that the calculus (with typed sets and) with terminal invention is $\mathcal{C}$-equivalent.

The calculus ^{ts}CALC extended with untyped sets, denoted CALC, is the calculus using rtypes instead of types. There is an intuitive correspondence between untyped sets and invented values in this context. Consider a query Q whose formula contains a quantified variable x of the rtype $\{Obj\}$. Then x ranges over arbitrarily large finite sets, all constructed from atomic objects appearing either in the input database instance or as constants in Q (if any). The elements in x can be used in the same manner as invented values.

We now informally review the semantics of ^{ts}CALC with invented values introduced in [HS88b]. For a query $Q \in {}^{ts}$CALC, a database d, and $i \in \mathbf{N}$, the semantics of Q under d with i invented values, denoted $Q|_i[d]$, is obtained as follows: (a) evaluate Q under limited interpretation with the active domain extended to include i new values (denote the answer $Q|^i[d]$); and (b) delete from $Q|^i[d]$ objects containing invented values. $Q|_\omega[d]$ and $Q|^\omega[d]$ are defined analogously. Note that the limited interpretation of Q is $Q|_0[d]$, and (assuming a countably infinite universal domain) the unlimited interpretation of Q is $Q|_\omega[d]$.

Definition: If $Q \in {}^{ts}$CALC, the semantics of Q under *finite invention*, Q^{fi}, is defined as $Q^{fi}[d] = \bigcup_{0 \le i < \omega} Q|_i[d]$ for all database instances d; the semantics of Q under *countable invention*, Q^{ci}, is defined as $Q^{ci}[d] = Q|_\omega[d]$ for all d. Let ^{ts}CALC^{fi} and ^{ts}CALC^{ci} denote the families of calculus queries with finite and countable invention semantics (respectively).

Theorem 6.1: [HS88b] ^{ts}CALC^{ci} is strictly more powerful than ^{ts}CALC^{fi}, which in turn is strictly more powerful than $\mathcal{C}$.

To illustrate the power of ^{ts}CALC^{ci} and ^{ts}CALC^{fi} we include:

Example 6.2: [HS88b]. Let M be a (conventional) Turing machine with unary input alphabet $\{a\}$; and let c be a constant in $\mathbf{U}$. Then there is a query Q in ^{ts}CALC^{fi} which computes the total function

$$f_{halt}(d) = \begin{cases} \{[c]\} & \text{if } M \text{ halts on } a^{|d|}; \\ \emptyset & \text{otherwise} \end{cases}$$

and there is a query Q' in ${}^{ts}\text{CALC}^{ci}$ which computes the query

$$f_{\overline{halt}}(d) = \{[c]\} - f_{halt}(d)$$

Intuitively, the body of Q outputs the tuple $\langle c \rangle$ if there exists a halting computation of M on the input $a^{|d|}$ whose running time is $\leq$ the number of active domain and invented objects. Because the semantics of Q is obtained by taking the union of its output on all finite sets of invented values, Q essentially has access to computations of M of all possible lengths. It is shown in [HS88b] that no query in ${}^{ts}\text{CALC}^{fi}$ can compute $f_{\overline{halt}}$. Under countable invention, Q' can simultaneously examine all possible computations of M on the input, and thus compute $f_{\overline{halt}}$. □

Let $\text{CALC}_\exists$ denote the class of calculus queries whose variables of rtypes which are not types are all existentially quantified.

Theorem 6.3: (a) $\text{CALC} \equiv {}^{ts}\text{CALC}^{ci}$; and (b) $\text{CALC}_\exists \sqsubseteq {}^{ts}\text{CALC}^{fi}$.

Proof: (sketch) (a) To show that ${}^{ts}\text{CALC}^{ci} \sqsubseteq \text{CALC}$, note that if a is a constant then the set $cons_{\mathbf{Obj}}(\{a\})$ is countably infinite; this can be used as the set of invented values. For the opposite direction, let Q be a query in CALC. The central problem in building a query Q' in ${}^{ts}\text{CALC}^{ci}$ which simulates Q is the removal of the type **Obj** wherever it occurs in Q. This is accomplished by "flattening" each element of $cons_{\mathbf{Obj}}(adom(d, Q))$ into an object of type $\{[U, U, U, U]\}$ which uses invented values. This flattening is reminiscent of the representation of complex objects used in the Logical Data Model [KV84]. (The proof of Lemma 6.5 in [HS88a] uses this technique of flattening in a slightly different context, and illustrates how it can be simulated in ${}^{ts}\text{CALC}$.)

(b) To simulate a query Q in $\text{CALC}_\exists$ by a query in ${}^{ts}\text{CALC}^{fi}$, first form $Q' \in \text{CALC}_\exists$ by moving all existentially quantified variables involving the type **Obj** to the outside. The body of Q' now has the form $\exists x_1/T_1 \ldots \exists x_n/T_n \psi$ where ψ has no quantified variables with types involving **Obj**. For each assignment α for $x_1, \ldots, x_n$, $\psi[\alpha]$ can be simulated using flattening and a finite set of invented values (whose size depends on α). Because the output of Q' is based on taking the union of the "outputs" of $\psi[\alpha]$ for each α, Q' can be simulated by the semantics of finite invention. □

Hence, a pure form of calculus with untyped sets (and complex object calculus with unlimited interpretation) is not practical under Church's thesis. We conclude by proposing the following new semantics for ${}^{ts}\text{CALC}$, which is shown to be $\mathcal{C}$-equivalent. (The full details of this development are presented in [HS88a].)

Definition: If $Q \in {}^{ts}\text{CALC}$, the semantics of Q under *terminal invention*, Q^{ti}, is:

$$Q^{ti}[d] = \begin{cases} Q|_n[d] & \text{if } n \text{ is least such that} \\ & Q|^n[d] \text{ contains an} \\ & \text{invented value,} \\ ? & \text{otherwise.} \end{cases}$$

The family of such queries is denoted as ${}^{ts}\text{CALC}^{ti}$.

Theorem 6.4: ${}^{ts}\text{CALC}^{ti}$ is $\mathcal{C}$-equivalent.

Proof: (sketch) It is obvious that each calculus query under terminal invention is computable and generic, hence in $\mathcal{C}$. Let f be a query function in $\mathcal{C}$ and M be a Turing machine which computes f. The construction of query $Q \in {}^{ts}\text{CALC}$ is essentially similar to that in the proof of Theorem 2.2. The only difference is that an arbitrarily large number of indices are now available under invention. Hence, we can use the type $\{[U, U, U, U]\}$ to store the computation of M. The rest of constructions are fairly direct. □

It should be noted that the results on expressive power of ${}^{ts}\text{CALC}$ with invented values rely on the presence of variables ranging over set types. For example, it was demonstrated in [AGSS86] and independently developed in [HS88b] (details appear in [HS89]) that the relational calculus with (finite or countable) invention has only the expressive power of the conventional relational calculus. In other words, the relational calculus with the unlimited interpretation has the same expressive power as with the limited interpretation.

7 Conclusion

The results described in this paper characterize the expressive power of query languages based on conventional paradigms from databases, but interpreted in a context where untyped (or unrestrictedly heterogeneous) sets are permitted. The languages studied cover a spectrum of expressive powers, ranging

from the algebra without **while**, which is equivalent to $\mathcal{E}$; through "most" langauges, which have the power of $\mathcal{C}$; up to the calculus, which has an expressive power strictly greater than $\mathcal{C}$.

While the technical discussion focused primarily on the use of untyped sets, analogous results hold in other cases where untyped sets can be simulated. Such situations include the use of list structures and the use of a freely interpreted function symbol (which has at least two input arguments). These constructs are typical of database programming languages such as, e.g., GALILEO [ACO85].

Acknowledgement

The notion of terminal invention was inspired by a conversation with Victor Vianu. We also thank Peter Buneman and Aaron Watters for helpful discussions concerning the relationship between natural join and BK.

References

[AB88] S. Abiteboul and C. Beeri. *On the Power of Languages for the Manipulation of Complex Objects.* Technical Report No.846, INRIA, May 1988.

[ACO85] A. Albano, L. Cardelli, and R. Orisini. Galileo: a strongly-typed, interactive conceptual language. *ACM Trans. on Database Systems*, 10(2):230–260, June 1985.

[AG87] S. Abiteboul and S. Grumbach. COL: a language for complex objects based on recursive rules (extended abstract). In *Proc. Workshop on Database Programming Languages*, pages 253 – 276, Roscoff, France, September 1987.

[AGSS86] A. K. Aylamazyan, M. M. Gilula, A. P. Stolboushkin, and G. F. Schwartz. Reduction of the relational model with infinite domain to the case of finite domains. *Proc. of USSR Acad. of Science (Doklady)*, 286(2):308–311, 1986.

[AV87] S. Abiteboul and V. Vianu. *Transaction Languages for Database Update and Specification.* Technical Report No.715, INRIA, September 1987.

[AV88] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proc. ACM Symp. on Principles of Database Systems*, pages 240–250, 1988.

[BBKV87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. Int. Conf. on Very Large Databases*, pages 97–105, 1987.

[BK86] F. Bancilhon and S. Khoshafian. A calculus for complex objects. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1986.

[BNR*87] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Schmueli, and S. Tsur. Sets and negation in a logic database language (LDL1). In *Proc. ACM Symp. on Principles of Database Systems*, 1987.

[CH80] A. K. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, Oct. 1980.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, 1984.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[GvG88] M. Gyssens and D. van Gucht. The powerset algebra as a result of adding programming constructs to the nested relational algebra. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1988.

[HK87] R. Hull and R. King. Semantic data modeling: survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[HS88a] R. Hull and J. Su. *On the Expressive Power of Database Queries with Intermediate Types.* Technical Report 88-53, Computer Science Department, Univ. of Southern California, 1988. Invited to special issue of *Journal of Computer and System Sciences.*

[HS88b] R. Hull and J. Su. On the expressive power of database queries with intermediate types. In *Proc. ACM Symp. on Principles of Database Systems*, pages 39–51, 1988.

[HS89] R. Hull and J. Su. *Domain Independence and the Relational Calculus.* Technical Report, Computer Science Dept, Univ of Southern California, 1989. In preparation.

[Hul87] R. Hull. A survey of theoretical research on typed complex database objects. In J. Paredaens, editor, *Databases*, pages 193–256, Academic Press (London), 1987.

[Kol87] P.G. Kolaitis. The expressive power of stratified logic programs. 1987. manuscript, Stanford University.

[KP88] P.G. Kolaitis and C.H. Papadimitriou. Why not negation by fixpoint? In *Proc. ACM Symp. on Principles of Database Systems*, 1988.

[KV84] G. M. Kuper and M. Y. Vardi. A new approach to database logic. In *Proc. ACM Symp. on Principles of Database Systems*, pages 86–96, 1984.

[KV88] G. M. Kuper and M. Y. Vardi. On the complexity of queries in the logical data model. In *Proc. Int. Conf. on Database Theory*, pages 267–280, 1988.

[PvG88] J. Paredaens and D. van Gucht. Possibilities and limitations of using flat operators in nested algebra expressions. In *Proc. ACM Symp. on Principles of Database Systems*, Austin, Texas, 1988.

[RKS85] M. A. Roth, H. F. Korth, and A. Silberschatz. *Extended algebra and calculus for not1NF Relational Databases.* Technical Report TR-85-19, University of Texas at Austin, 1985.