NORMAL FORMS AND CONSERVATIVE EXTENSION PROPERTIES FOR QUERY LANGUAGES OVER COLLECTION TYPES*

Limsoon Wong

Real World Computing Partnership Novel Function
Institute of Systems Science Laboratory
Heng Mui Keng Terrace, Singapore 0511

18 July 1995

*This reviewed invited paper appeared in *Journal of Computer and System Sciences*, 52(3):495–505, June 1996.

CONSERVATIVE EXTENSION PROPERTY

Correspondence to: Limsoon Wong, Institute of Systems Science, Heng Mui Keng Terrace, Singapore 0511. Email: limsoon@saul.cis.upenn.edu.

Abstract

Strong normalization results are obtained for a general language for collection types. An induced normal form for sets and bags is then used to show that the class of functions whose input has height (that is, the maximal depth of nestings of sets/bags/lists in the complex object) at most i and output has height at most o definable in a nested relational query language without *powerset* operator is *independent* of the height of intermediate expressions used. Our proof holds regardless of whether the language is used for querying sets, bags, or lists, even in the presence of variant types. Moreover, the normal forms are useful in a general approach to query optimization. Paredaens and Van Gucht proved a similar result for the special case when $i = o = 1$. Their result is complemented by Hull and Su who demonstrated the failure of independence when *powerset* operator is present and $i = o = 1$. The theorem of Hull and Su was generalized to all i and o by Grumbach and Vianu. Our result generalizes Paredaens and Van Gucht's to all i and o, providing a counterpart to the theorem of Grumbach and Vianu.

1 Introduction

In Breazu-Tannen, Buneman, and Wong [7], a nested relational calculus and a nested relational algebra based on structural recursion [6, 5] and on monads [34, 22] were proposed. In this report, we describe *relative set abstraction* as a third nested relational query language. This query language is similar to the well-known list comprehension mechanism in functional programming languages such as Miranda [31] and KRC [30]. This language is equivalent to the two earlier query languages both in terms of semantics and in terms of equational theories. This strong sense of equivalence allows these three query languages to be freely combined into a nested relational query language, called $\mathcal{NRL}$, and allows one to prove many properties about it by looking only at one of the original languages.

In particular, we show that every expression of relative set abstraction can be reduced to a normal form. This normal form has an immediately apparent property: *an expression in normal form does not have any subexpression with set height exceeding the set height of the type of the expression.* Here, the set height of a complex object refers to the maximal depth of nesting of sets in that object, and similarly for object types. For functions from objects to objects, the set height of the function type is the maximum of the set height of the input and output types. Let $\mathcal{NRL}_{i,o,k}$ denote the class of functions whose input has set height at most i and whose output has set height at most o and are definable in $\mathcal{NRL}$ using intermediate expressions whose set heights are at most $k \geq \max(i, o)$. Then this result says that for any i, o, $k \geq \max(i, o)$, $\mathcal{NRL}_{i,o,k}$ coincides with $\mathcal{NRL}_{i,o,k+1}$ as a class. In other words, $\mathcal{NRL}_{i,o,k+1}$ is a *conservative extension* of $\mathcal{NRL}_{i,o,k}$ as a language. Consequently, the class $\mathcal{NRL}_{i,o,k}$ is *independent* of k. Thus the ability to use intermediate expressions of great height does not increase expressive power.

As an example of the conservative extension property, let us consider several possible ways to test whether every drinker likes the same selection of beers. Let $R : \{drinker \times beer\}$ tabulate which drinker likes what beers. One way is to first group by the beers around each invidual drinker and then test whether these groups are all identical. Another way to express the same query is to test whether the cartesian product of all drinkers and beers in R is equal to R itself. The first method results in an intermediate set having one extra level of nesting — the set containing the groups of beers. On the other hand, the second method needs nothing more than flat relations. Having the conservative extension property means that any query, such as the drinker-and-beer problem above, that is expressible using some deeply-nested intermediate data, such as the first method above, can always be expressed using intermediate data that is less deeply nested, such as the second method above. A third method is to produce the groups of beers liked by each individual drinker one at a time and perform an on-the-fly test to see if the current group contains all the beers mentioned in R. This last method corresponds to the optimization idea known as pipelining and it also does not need nested relations. All three methods can be expressed as queries in $\mathcal{NRL}$. The conservative extension property in this paper is essentially proved by showing that queries like the first method can always be optimized into queries like the third method, if $\mathcal{NRL}$ is the query language.

This research complements work by other researchers. To begin with, Paredaens and Van Gucht [25] showed that the nested relational algebra of Thomas and Fischer [27] is conservative with respect to flat relational algebra in the sense we have described. Since the language of Thomas and Fischer is equivalent to ours [36], this result implies that $\mathcal{NRL}_{i,o,k+1}$ is conservative with respect to $\mathcal{NRL}_{i,o,k}$ when $i = o = 1$. Our result generalizes this to conservativeness for all i and o. Hull and Su proposed a nested relational query language in which *powerset* is expressible and studied its expressive power [13]. One of their results is that it is not conservative with respect to the flat relational algebra in this sense. Adding the *powerset* operator to $\mathcal{NRL}$ gives us a language equivalent to Hull and Su's. Hence $\mathcal{NRL}(powerset)_{i,o,k+1}$ is not conservative with respect to $\mathcal{NRL}(powerset)_{i,o,k}$ when $i = o = 1$. Grumbach and Vianu [10] proved that the language of Hull and Su is not conservative with respect to set height of input/output at all, implying the failure of conservativeness in $\mathcal{NRL}(powerset)$ for all i and o. In contrast, our language cannot express *powerset* and is conservative with respect to set height of input/output.

The general conservative extension result can be further improved in two ways. Firstly, many modern data models possess an additional data structuring mechanism known variously as co-products, variant types, sum types, or tagged unions; see Abiteboul and Hull [3] and Hull and Yap [14]. However, many papers on expressive power do not consider this feature [13, 10, 2]. We extend the nested relational calculus of Breazu-Tannen, Buneman, and Wong [7] with variant types and prove that the extended calculus remains conservative with respect to height of input/output.

Secondly, the proof we give for relative set abstraction relies on a set-based semantics. This is in line with the work of many researchers as reported in Abiteboul et. al. [1], Abiteboul and Beeri [2], Hull and Su [13], Grumbach and Vianu [10], Paredaens and Van Gucht [25], and Gyssens and Van Gucht [12]. But our languages can also be given interpretations based on bags and lists. It is desirable to know whether the main result holds when the languages are used to manipulate nested lists and bags. We prove that it does. Moreover, the proof is *uniform* across these semantics.

The organization of this paper is as follows. Section 2 introduces relative set abstraction and the nested relational calculus of Breazu-Tannen, Buneman, and Wong [7]. We establish translations between these languages that preserve semantics, preserve set heights, and preserve and reflect equational theories. Section 3 presents a strongly normalizing rewrite system. It is then used to show the main result that the query language is conservative with respect to set height of input/output. The two improvements mentioned above are presented in the Section 4.

2 Relative Set Abstraction

First let us sketch the calculus of Breazu-Tannen, Buneman, and Wong [7] (or $\mathcal{NRC}$ for short). Note that they simulated the Booleans using $\{()\}$ and $\{\}$ for reason of conceptual economy. In this paper, we use real Booleans for reason of readability.

Types. A type in $\mathcal{NRC}$ is either an object type s or is a function type $s \to t$ where s and t are both object types. The object types are given by the grammar:

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

The semantic of a complex object type is just a set of complex objects. The type *unit* has precisely one object which we denote $()$. The type *bool* has as objects the two Boolean values, *true* and *false*. There are also some unspecified base types b. An object of type $s \times t$ is a pair whose first component is an object of type s and whose second component is an object of type t. An object of type $\{s\}$ is a finite set whose elements are objects of type s.

Expressions. The expressions of $\mathcal{NRC}$ are formed according to the rules in Figure 1. Type superscripts are usually omitted because they can be inferred [21]. In fact, they remain inferrable even when records instead of pairs are used. See Ohori [24]; Ohori, Buneman, and Breazu-Tannen [23]; Jategaonkar and Mitchell [16]; and Remy [26]. The usual Barendregt convention [4] that bound variables are all distinct is adopted.

$$\frac{}{x^s : s} \qquad \frac{e : t}{\lambda x^s.e : s \to t} \qquad \frac{e_1 : s \to t \qquad e_2 : s}{e_1\ e_2 : t}$$

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

$$\frac{}{true : bool} \qquad \frac{}{false : bool} \qquad \frac{e_1 : bool \qquad e_2 : s \qquad e_3 : s}{if\ e_1\ then\ e_2\ else\ e_3 : s}$$

$$\frac{}{\{\}^s : \{s\}} \qquad \frac{e : s}{\{e\} : \{s\}} \qquad \frac{e_1 : \{s\} \qquad e_2 : \{s\}}{e_1 \cup e_2 : \{s\}} \qquad \frac{e_1 : \{t\} \qquad e_2 : \{s\}}{\bigcup\{e_1 \mid x^s \in e_2\} : \{t\}}$$

$$\frac{}{c : b} \qquad \frac{e_1 : b \qquad e_2 : b}{e_1 =^b e_2 : bool} \qquad \frac{e : \{unit\}}{empty\ e : bool}$$

Figure 1: Expressions of $\mathcal{NRC}$.

For example, $\bigcup\{\{\pi_1\ x^{s\times t}\} \mid x^{s\times t} \in R^{\{s\times t\}}\}$ is a valid expression because it can be typed, as shown below, according to our formation rules. On the other hand, $\bigcup\{\{x^s\} \mid x^s \in R^{int}\}$ is not a valid expression no matter what s is, because the type *int* is not a set type and thus cannot be formed according to our rules.

$$\dfrac{\dfrac{\dfrac{x^{s\times t} : s \times t}{\pi_1\ x^{s\times t} : s}}{\{\pi_1\ x^{s\times t}\} : \{s\}} \qquad \dfrac{}{R^{\{s\times t\}} : \{s \times t\}}}{\bigcup\{\{\pi_1\ x^{s\times t}\} \mid x^{s\times t} \in R^{\{s\times t\}}\} : \{s\}}$$

Semantics. The semantics of these constructs are described below. The expression x is used to denote the input object. The expression $\lambda x.e$ denotes the function f such that $f(x) = e$. The expression $e_1\ e_2$ denotes the result of applying the function e_1 to the object e_2.

It has already been mentioned that $()$ denotes the unique object of type *unit*. The expression (e_1, e_2) denotes the pair whose first component is the object denoted by e_1 and whose second component is the object denoted by e_2. The expression $\pi_1\ e$ denotes the first component of the pair denoted by e. The expression $\pi_2\ e$ denotes the second component of the pair denoted by e.

The expression $\{\}$ denotes the empty set. The expression $\{e\}$ denotes the singleton set containing the object denoted by e. The expression $e_1 \cup e_2$ denotes the union of the sets e_1 and e_2. The expression $\bigcup\{e_1 \mid x \in e_2\}$ denotes the set obtained by first applying the function $\lambda x.e_1$ to each object in the set e_2 and then taking the union of the results (which must be sets by the typing rule for the construct). That is, $\bigcup\{e_1 \mid x \in e_2\} = f(o_1) \cup \ldots \cup f(o_n)$, where f is the function denoted by $\lambda x.e_1$ and $\{o_1, \ldots, o_n\}$ is the set denoted by e_2. It should be emphasized that the $x \in e_2$ part in the $\bigcup\{e_1 \mid x \in e_2\}$ construct is not a membership test. It is an abstraction which introduces the variable x whose scope is the expression e_1; and it should be understood in the same spirit in which the lambda abstraction $\lambda y.e$ is understood.

The expressions *true* and *false* denotes the two Boolean values in the obvious way. The expression *if* e_1 *then* e_2 *else* e_3 has the usual meaning. That is, if e_1 is true, then the whole expression denotes the same object as e_2; and if e_1 is false, then the whole expression denotes the same object as e_3.

The expression c denotes a constant of base type b. The expression $e_1 =^b e_2$ is the equality test restricted to base type b. As will be shown later, the equality tests at all complex object types are definable in terms of $=^b$ using $\mathcal{NRC}$ as the ambient language. Finally, the expression *empty* e is the emptiness test restricted to the *unit* type. Testing for emptiness of sets e' at other types can be expressed as $\mathit{empty}\{() \mid x \in e'\}$.

Shorthands. The following shorthands are very intuitive and we use them whenever possible. The "expression" *not* e is to be interpreted as *if* e *then false else true*. The "expression" e_1 *and* e_2 is to be interpreted as *if* e_1 *then* e_2 *else false*.

Examples. Let X and Y denote sets having types $\{s \times \{t\}\}$ and $\{t'\}$ respectively. Then $\bigcup\{\bigcup\{\{(x, y)\} \mid x \in X\} \mid y \in Y\}$ has type $\{(s \times \{t\}) \times t'\}$ and denotes the cartesian product of the sets denoted by X and Y; while $\bigcup\{\bigcup\{\{(\pi_1 x, y)\} \mid y \in \pi_2 x\} \mid x \in X\}$ has type $\{s \times t\}$ and denotes the unnesting of the set denoted by X.

Wadler and Trinder argued that list/set/bag comprehensions is a natural query notation [29, 28, 35]. They also demonstrated that this notation does not hamper query optimization. In the remainder of this section we present a query language based on the comprehension syntax that is equivalent to $\mathcal{NRC}$. We call this query language *Relative Set Abstraction* (or $\mathcal{RSA}$ for short).

Types. The types in $\mathcal{RSA}$ are the same as those in $\mathcal{NRC}$.

Expressions. These are the same as $\mathcal{NRC}$, but with the $\bigcup\{e \mid x \in e'\}$ construct replaced by the set comprehension construct $\{e \mid x_1 \in e_1, \ldots, x_n \in e_n\}$ whose typing rule is given in Figure 2. Note that the e in the comprehension construct is not required to be a set.

$$\frac{e_1 : \{s_1\} \qquad \ldots \qquad e_n : \{s_n\} \qquad e : t}{\{e \mid x_1^{s_1} \in e_1, \ldots, x_n^{s_n} \in e_n\} : \{t\}}$$

Figure 2: Set comprehension in $\mathcal{RSA}$.

The lexical ordering of $x_1 \in e_1, \ldots, x_n \in e_n$ in $\{e \mid x_1 \in e_1, \ldots, x_n \in e_n\}$ is significant, since x_i can be used in e_j, for $j > i$. It must be pointed out that, as in the $\bigcup\{e_1 \mid x \in e_2\}$ construct, the $x_i \in e_i$ in the comprehension construct is *not* a set membership test. It is the introduction of a variable binding, similar to that of lambda abstraction $\lambda x.e$. It is to emphasize this point that we call this language *relative set abstraction*.

We use the notation Δ as a shorthand for $x_1 \in e_1, \ldots, x_n \in e_n$. The scope of a set abstraction variable x_i in $\{e \mid \Delta, x_i \in e_i, \Delta'\}$ is Δ' and e. There is a close syntactic similarity between $\mathcal{RSA}$ and the traditional flat relational calculus. In fact, the lexical ordering constraint mentioned above can be seen as a straightforward device for guaranteeing safety. This simple constraint, though apparently more restrictive than those safety constraints imposed on relational calculus, does not lead to a loss in expressive power.

Semantics. The meaning of $\{e \mid x_1 \in e_1, \ldots, x_n \in e_n\}$ is the set $f(o_1) \cup \ldots \cup f(o_m)$, where f is the function such that $f(x_1) = \{e \mid x_2 \in e_2, \ldots, x_n \in e_n\}$ and $\{o_1, \ldots, o_m\}$ is the set e_1. For the base case, the meaning of $\{e \mid \}$ is just the singleton set $\{e\}$. Thus, the semantics can be defined in terms of $\mathcal{NRC}$ as follows:

$$\{e \mid x_1 \in e_1,\ \Delta\} = \bigcup\{\{e \mid \Delta\} \mid x_1 \in e_1\}$$

This, with the semantics of the base case, provides a recursive definition of comprehensions purely in terms of the $\bigcup\{e_1 \mid x \in e_2\}$ construct.

Shorthand. The following shorthand is very intuitive and we use it whenever possible. The "expression" $\{e \mid \Delta_1,\ e', \Delta_2\}$, where e' has type *bool*, is to be interpreted as $\{e \mid \Delta_1,\ x \in (\textit{if } e' \textit{ then } \{()\} \textit{ else } \{\}),\ \Delta_2\}$, where x is a fresh variable.

Examples. Let X and Y denote sets having types $\{\{s\}\}$ and $\{t\}$ respectively. Then $\{(x, y) \mid x \in X,\ y \in Y\}$ has type $\{\{s\} \times t\}$ and denotes the cartesian product of the sets denoted by X and Y. As a second example, $\{y \mid x \in X,\ y \in x\}$ has type $\{s\}$ and denotes the flattening of the set denoted by X. For a more ambitious example, let W denote a set having type $\{s \times t\}$. Then $\{(\pi_1 x,\ \{\pi_2 y \mid y \in W,\ \pi_1 y = \pi_1\ x\}) \mid x \in W\}$ has type $\{s \times \{t\}\}$ and denotes the nesting operation on the first column of W.

Having introduced the languages, we show that they are equivalent. To this end, we need a translation $\mathcal{NR}[\cdot]$ taking an expression $e : s$ of $\mathcal{NRC}$ to an expression $\mathcal{NR}[e] : s$ of $\mathcal{RSA}$ and a translation $\mathcal{RN}[\cdot]$ taking an expression $e : s$ of $\mathcal{RSA}$ to an expression $\mathcal{RN}[e] : s$ of $\mathcal{NRC}$. The translations are straightforward [34]. Since the languages differ only in one pair of constructs, only rules for this pair are needed.

- $\mathcal{NR}[\bigcup\{e_1 \mid x \in e_2\}] = \{y \mid x \in \mathcal{NR}[e_2],\ y \in \mathcal{NR}[e_1]\}$, where y is fresh.
- $\mathcal{RN}[\{e \mid x_1 \in e_1, \ldots, x_n \in e_n\}] = \bigcup\{\mathcal{RN}[\{e \mid x_2 \in e_2, \ldots, x_n \in e_n\}] \mid x_1 \in \mathcal{RN}[e_1]\} = \bigcup\{\ldots \bigcup\{\{\mathcal{RN}[e]\} \mid x_n \in \mathcal{RN}[e_n]\} \mid \ \ldots\} \mid x_1 \in \mathcal{RN}[e_1]\}$.

Theorem 2.1 *Every closed e of $\mathcal{NRC}$ denotes the same value as $\mathcal{NR}[e]$; and every closed e of $\mathcal{RSA}$ denotes the same value as $\mathcal{RN}[e]$. That is, $\mathcal{RSA}$ and $\mathcal{NRC}$ are equivalent.* $\square$

Since $\mathcal{NRC}$ does not have membership test, or anything that looks like a nesting operation, we show that they are definable. It has been established [7] that equality test $=^s$ on all object types s can be used to simulate membership test, subset test, set difference, set intersection, and relational nesting. Therefore, it suffices to prove that

Proposition 2.2 *Equality at all complex object types is definable in $\mathcal{RSA}$.*

Proof. Let $=^s$ be the equality test at type s. It can be defined by induction on s.

- $x \ =^{unit} y \ = \ true$
- $x =^{bool} y \ = \ if\ x\ then\ y\ else\ (not\ y)$
- $x \ =^{b} y$ is given.
- $x \ =^{s \times t} \ y \ = \ (\pi_1 x \ =^s \pi_1 y)\ and\ (\pi_2 x =^t \pi_2 y)$
- $X \ =^{\{s\}} \ Y \ = \ (X\ subset^s\ Y)\ and\ (Y\ subset^s\ X)$, where
- $X\ subset^s\ Y \ = \ empty\{() \mid x \in X,\ (x\ nonmember^s Y)\}$
- $x\ nonmember^s\ Y \ = \ empty\{() \mid y \in Y, x =^s y\}$. $\square$

3 Normal Form and Conservativity

We first present a rewrite system for $\mathcal{RSA}$ that is strongly normalizing. The normal forms induced by this rewrite system are then used to prove that every definable function is definable using subexpressions whose set height is at most the set height of the input/output of the function. The *set height* $ht(s)$ of a type s is defined by induction on the structure of type:

- $ht(unit) = ht(bool) = ht(b) = 0$
- $ht(s \times t) = ht(s \to t) = \max\{ht(s), ht(t)\}$
- $ht(\{s\}) = 1 + ht(s)$

Note that every expression of our languages has a unique typing derivation. The set height of expression e is defined simply as $ht(e) = \max\{ht(s) \mid s \text{ occurs in the type derivation of } e\}$. Then the theorem expresses a very general conservative property. It says that to process information (that is, input/output) of set height n, no operators whose set height exceed n is required. In other words, if a function whose input/output has height n is defined by an expression e whose height exceeds n, we can find an alternative expression e' whose height does not exceed n to implement it.

As an illustration, let us consider the first method mentioned earlier for testing if all drinkers like the same beers. It can be implemented by the expression e defined as $empty\{() \mid z \in \{\{\pi_2 y \mid y \in R,\ (\pi_1 y =^{drinker} \pi_1 x)\} \mid x \in R\},\ not\ (z =^{\{beer\}} \{\pi_2 w \mid w \in R\})\}$, where $R : \{drinker \times beer\}$ tabulates which drinker likes what beers. This expression has height 2 because the subexpression $\{\{\pi_2 y \mid y \in R,\ (\pi_1 y =^{drinker} \pi_1 x)\} \mid x \in R\}$ has height 2. This expression e having the sole free variable R implicitly defines a function $f(R) = e$. (This function f can also be formally defined as $\lambda R.e$ in $\mathcal{RSA}$.) Thus, the value of R is the input to f and the value of e given R is the output of f. So f has input height $1 = ht(R)$, output height $0 = ht(bool)$, and set height $2 = ht(e)$.

In other words, the function f uses intermediate data that has a greater level of set nesting than its input and output. We now introduce a rewrite system to eliminate this problem by deriving a flat implementation of f. Let $e[e'/x]$ stands for the expression obtained by replacing all free occurrences of x in e by e', provided the free variables in e' are not captured during the substitution. Similarly, the notation $\Delta[e'/x]$, where Δ is $x_1 \in e_1, \ldots, x_n \in e_n$, stands for $x_1 \in e_1[e'/x], \ldots, x_n \in e_n[e'/x]$. Now, consider the rewrite system consisting of the following rules:

1. $(\lambda x.e)e' \rightsquigarrow e[e'/x]$
2. $\pi_i(e_1, e_2) \rightsquigarrow e_i$
3. $if\ true\ then\ e_1\ else\ e_2 \rightsquigarrow e_1$

4. *if false then* e_1 *else* $e_2 \leadsto e_2$
5. *if* (*if* e_1 *then* e_2 *else* e_3) *then* e_4 *else* $e_5 \leadsto$ *if* e_1 *then* (*if* e_2 *then* e_4 *else* e_5) *else* (*if* e_3 *then* e_4 *else* e_5)
6. $\pi_i($*if* e_1 *then* e_2 *else* $e_3) \leadsto$ *if* e_1 *then* $\pi_i\ e_2$ *else* $\pi_i\ e_3$
7. $\{e \mid \Delta_1,\ x \in \{\},\ \Delta_2\} \leadsto \{\}$
8. $\{e \mid \Delta_1,\ x \in \{e'\},\ \Delta_2\} \leadsto \{e[e'/x] \mid \Delta_1,\ \Delta_2[e'/x]\}$
9. $\{e \mid \Delta_1,\ x \in e_1 \cup e_2,\ \Delta_2\} \leadsto \{e \mid \Delta_1,\ x \in e_1,\ \Delta_2\} \cup \{e \mid \Delta_1,\ x \in e_2,\ \Delta_2\}$
10. $\{e \mid \Delta_1,\ x \in \{e' \mid \Delta'\},\ \Delta_2\} \leadsto \{e[e'/x] \mid \Delta_1,\ \Delta',\ \Delta_2[e'/x]\}$
11. $\{e \mid \Delta_1,\ x \in$ *if* e_1 *then* e_2 *else* $e_3,\ \Delta_2\} \leadsto \{e \mid \Delta_1,\ u \in$ *if* e_1 *then* $\{()\}$ *else* $\{\},\ x \in e_2,\ \Delta_2\} \cup \{e \mid \Delta_1,\ u \in$ *if* e_1 *then* $\{\}$ *else* $\{()\},\ e_3,\ \Delta_2\}$, provided (1) u is fresh, (2) e_2 is not $\{()\}$ and e_3 is not $\{\}$, and (3) e_2 is not $\{\}$ and e_3 is not $\{()\}$.

Rule 10 is the most significant rule. It rewrites the expression $\{e \mid \Delta_1,\ x \in \{e' \mid \Delta'\},\ \Delta_2\}$ to $\{e[e'/x] \mid \Delta_1,\ \Delta',\ \Delta_2[e'/x]\}$. In the process of doing so, it eliminates the intermediate set $\{e' \mid \Delta'\}$ constructed by the original expression. If this intermediate set has great height, than the set height of the resulting expression would be reduced.

Rule 11 basically rewrites $\{e \mid \Delta_1,\ x \in$ *if* e_1 *then* e_2 *else* $e_3,\ \Delta_2\}$ to $\{e \mid \Delta_1,\ e_1,\ x \in e_2,\ \Delta_2\} \cup \{e \mid \Delta_1,$ *not* $e_1,\ x \in e_3,\ \Delta_2\}$. It is given the more complicated form above in order to guarantee the termination of the system. In the next section, we present a strikingly simpler system based on $\mathcal{NRC}$.

Rule 5 is not really needed for proving the conservative extension theorem in this section. It is included here to provide a correspondence to a more general rule used in proving the more general result of the next section. It is of course also a useful simplification rule in its own right.

As an illustration of these rules, let us consider the first method for testing if all drinkers like the same beers: *empty*$\{() \mid z \in \{\{\pi_2 y \mid y \in R,\ (\pi_1 y =^{drinker} \pi_1 x)\} \mid x \in R\},$ *not* $(z =^{\{beer\}} \{\pi_2 w \mid w \in R\})\}$. As discussed earlier, it has set height 2. It can be rewritten using Rule 10 to give the expression *empty* $\{() \mid x \in R,$ *not* $(\{\pi_2 y \mid y \in R,\ (\pi_1 y =^{drinker} \pi_1 x)\} =^{\{beer\}} \{\pi_2 w \mid w \in R\})\}$, which has height 1 and is the third method mentioned earlier. The difference between these two expressions is simple. The original expression generates all the grouping of beers $\{\{\pi_2 y \mid y \in R,\ (\pi_1 y =^{drinker} \pi_1 x)\} \mid x \in R\}$ before testing that each group $\{\pi_2 y \mid y \in R,\ (\pi_1 y =^{drinker} \pi_1 x)\}$ is the same as all the beers mentioned in R. The new expression generates one group $\{\pi_2 y \mid y \in R,\ (\pi_1 y =^{drinker} \pi_1 x)\}$ and tests it before going on to the next group, avoiding the need to keep all groups simultaneously. Note that the expression can be further reduced because $=^{\{beer\}}$ is a compound expression defined in terms of $=^{beer}$ as given by Proposition 2.2. However, these subsequent rewrite steps do not change the height of expressions.

This rewrite system is sound. That is,

Proposition 3.1 (Soundness) *If $e_1 \leadsto e_2$, then e_1 and e_2 have the same denotation.* □

A rewrite system is said to be strongly normalizing if it does not admit any infinite sequence of rewriting. That is, any sequence of rewriting must lead to an expression to which no rewrite rule of the system is applicable.

Theorem 3.2 (Strong normalization) *This rewrite system is strongly normalizing.*

Proof. Let φ be an arbitrary function which maps variable names to a natural number greater than 1 and let $\varphi[n/x]$ be the function which assigns n to x but agrees with φ on other variables. Then $\|e\|\varphi$, as defined below, measures the size of e in the environment φ where each free variable x in e is given the size $\varphi(x)$.

- $\|x\|\varphi = \varphi(x)$
- $\|()\|\varphi = \|\{\}\|\varphi = \|\mathit{true}\|\varphi = \|\mathit{false}\|\varphi = \|c\|\varphi = 2$
- $\|\pi_1\ e\|\varphi = \|\pi_2\ e\|\varphi = \|\{e\}\|\varphi = \|\mathit{empty}\ e\|\varphi = 1 + \|e\|\varphi$
- $\|(\lambda x.e)(e')\|\varphi = \|e\|\varphi[\|e'\|\varphi/x] + \|e'\|\varphi$
- $\|\lambda x.e\|\varphi = \|e\|\varphi[2/x]$
- $\|\mathit{if}\ e_1\ \mathit{then}\ \{()\}\ \mathit{else}\ \{\}\|\varphi = \|\mathit{if}\ e_1\ \mathit{then}\ \{\}\ \mathit{else}\ \{()\}\|\varphi = \|e_1\|\varphi$
- $\|\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3\|\varphi = \|e_1\|\varphi \cdot (1 + \|e_2\|\varphi + \|e_3\|\varphi)$, provided the cost formula immediately above is not applicable.
- $\|e_1\ \cup\ e_2\|\varphi = \|(e_1, e_2)\|\varphi = \|e_1\ =^b\ e_2\|\varphi = 1 + \|e_1\|\varphi + \|e_2\|\varphi$
- $\|\{e_n \mid x_1 \in e_0, \ldots, x_n \in e_{n-1}\}\|\varphi = \|e_0\|\varphi_0 \cdot \ldots \cdot \|e_n\|\varphi_n$, where $\varphi_0 = \varphi$ and $\varphi_{i+1} = \varphi_i[\|e_i\|\varphi_i/x_{i+1}]$.

Now we need several technical claims.

Claim I. Suppose x is not free in e. Then $\|e\|\varphi = \|e\|\varphi[n/x]$.

Proof of Claim I. Straightforward induction on e.

Claim II. Suppose $\varphi_1(x) \leq \varphi_2(x)$ for each x free in e. Then $\|e\|\varphi_1 \leq \|e\|\varphi_2$.

Proof of Claim II. Straightforward induction on e.

Claim III. Suppose $\|e'\|\varphi \leq n$ and x not free in e'. Then $\|e[e'/x]\|\varphi \leq \|e\|\varphi[n/x]$.

Proof of Claim III. Since x is not free in e', it is also not free in $e[e'/x]$. By Claim I, it suffices to prove $\|e[e'/x]\|\varphi[n/x] \leq \|e\|\varphi[n/x]$ instead. This is easily accomplished by induction on e.

Claim IV. Suppose x is not free in e'. Suppose $\|e'\|\varphi_1 \leq \varphi_2(x)$. Suppose $\varphi_1(y) \leq \varphi_2(y)$ for each y, distinct from x, free in e. Then $\|e[e'/x]\|\varphi_1 \leq \|e\|\varphi_2$.

Proof of Claim IV. By Claim III, $\|e[e'/x]\|\varphi_1 \leq \|e\|\varphi_1[\varphi_2(x)/x]$. Clearly, $\varphi_1[\varphi_2(x)/x](y) \leq \varphi_2(y)$ for all y free in e. By Claim II, $\|e\|\varphi_1[\varphi_2(x)/x] \leq \|e\|\varphi_2$. Thus $\|e[e'/x]\|\varphi_1 \leq \|e\|\varphi_2$.

Claim V. Suppose $e_1 \leadsto e_2$. Then $\|e_1\|\varphi > \|e_2\|\varphi$ for any φ.

Proof of Claim V. With Claim II and Claim IV in our possession, the proof is a routine analysis on $e_1 \leadsto e_2$. We provide the two most interesting cases for illustration.

Case $\{e_k \mid x_1 \in e_0, \ldots, x_k \in e_{k-1}\} \leadsto \{e'_k \mid x_1 \in e'_0, \ldots, x_k \in e'_{k-1}\} \cup \{e''_k \mid x_1 \in e''_0, \ldots, x_k \in e''_{k-1}\}$, where e_n is $e'_n \cup e''_n$ for a certain fixed $n < k$; and e_i, e'_i, and e''_i are identical for $i \neq n$. Let $\varphi_0 = \varphi$ and $\varphi_{i+1} = \varphi_i[\|e_i\|\varphi_i/x_{i+1}]$. Let $\varphi'_0 = \varphi$ and $\varphi'_{i+1} = \varphi'_i[\|e'_i\|\varphi'_i/x_{i+1}]$. Let $\varphi''_0 = \varphi$ and $\varphi''_{i+1} = \varphi''_i[\|e''_i\|\varphi''_i/x_{i+1}]$. Then we calculate

$$\begin{array}{ll}
\|\{e_k \mid x_1 \in e_0, \ldots, x_k \in e_{k-1}\}\|\varphi & \\
= \|e_0\|\varphi_0 \cdot \ldots \cdot \|e_k\|\varphi_k & \\
= (\|e_0\|\varphi_0 \cdot \ldots \cdot \|e_{n-1}\|\varphi_{n-1}) \cdot (1 + \|e'_n\|\varphi_n + \|e''_n\|\varphi_n) \cdot (\|e_{n+1}\|\varphi_{n+1} \cdot \ldots \cdot \|e_k\|\varphi_k) & \\
> 1 + (\|e_0\|\varphi_0 \cdot \ldots \cdot \|e'_n\|\varphi_n \cdot \ldots \cdot \|e_k\|\varphi_k) + (\|e_0\|\varphi_0 \cdot \ldots \cdot \|e''_n\|\varphi_n \cdot \ldots \cdot \|e_k\|\varphi_k) & \\
\geq 1 + (\|e'_0\|\varphi'_0 \cdot \ldots \cdot \|e'_n\|\varphi'_n \cdot \ldots \cdot \|e'_k\|\varphi'_k) + (\|e''_0\|\varphi''_0 \cdot \ldots \cdot \|e''_n\|\varphi''_n \cdot \ldots \cdot \|e''_k\|\varphi''_k) & \text{By II.} \\
= \|\{e'_k \mid x_1 \in e'_0, \ldots, x_k \in e'_{k-1}\} \cup \{e''_k \mid x_1 \in e''_0, \ldots, x_k \in e''_{k-1}\}\|\varphi &
\end{array}$$

Case $\{e_k \mid x_1 \in e_0, \ldots, x_k \in e_{k-1}\} \leadsto \{e'_k \mid x_1 \in e_0, \ldots, x_n \in e_{n-1}, y_1 \in e''_0, \ldots, y_m \in e''_{m-1}, x_{n+2} \in e'_{n+1}, \ldots, x_k \in e'_{k-1}\}$, where e_n is $\{e''_m \mid y_1 \in e''_0, \ldots, y_m \in e''_{m-1}\}$ for a certain fixed $n < k$; and e'_i is $e_i[e''_m/x_{n+1}]$ for $i > n$. Let $\varphi_0 = \varphi$ and $\varphi_{i+1} = \varphi_i[\|e_i\|\varphi_i/x_{i+1}]$. Let $\varphi''_0 = \varphi_n$ and $\varphi''_{i+1} = \varphi''_i[\|e''_i\|\varphi''_i/y_{i+1}]$. Let $\varphi'_{n+1} = \varphi''_{m-1}[\|e''_{m-1}\|\varphi''_{m-1}/y_m]$ and $\varphi'_{i+1} = \varphi'_i[\|e'_i\|\varphi'_i/x_{i+1}]$. Then we calculate as follows:

$$\begin{array}{ll}
\|\{e_k \mid x_1 \in e_0, \ldots, x_k \in e_{k-1}\}\|\varphi & \\
= \|e_0\|\varphi_0 \cdot \ldots \cdot \|e_k\|\varphi_k & \\
= \|e_0\|\varphi_0 \cdot \ldots \|e_{n-1}\|\varphi_{n-1} \cdot \|e''_0\|\varphi''_0 \cdot \ldots \cdot \|e''_m\|\varphi''_m \cdot \|e_{n+1}\|\varphi_{n+1} \ldots \cdot \|e_k\|\varphi_k & \\
> \|e_0\|\varphi_0 \cdot \ldots \|e_{n-1}\|\varphi_{n-1} \cdot \|e''_0\|\varphi''_0 \cdot \ldots \cdot \|e''_{m-1}\|\varphi''_{m-1} \cdot \|e_{n+1}\|\varphi_{n+1} \ldots \cdot \|e_k\|\varphi_k & \\
\geq \|e_0\|\varphi_0 \cdot \ldots \|e_{n-1}\|\varphi_{n-1} \cdot \|e''_1\|\varphi''_1 \cdot \ldots \cdot \|e''_{m-1}\|\varphi''_{m-1} \cdot \|e'_{n+1}\|\varphi'_{n+1} \ldots \cdot \|e'_k\|\varphi'_k & \text{By IV.} \\
= \|\{e'_k \mid x_1 \in e_0, \ldots, x_n \in e_{n-1}, y_1 \in e''_0, \ldots, y_m \in e''_{m-1}, x_{n+2} \in e'_{n+1}, \ldots, x_k \in e'_{k-1}\}\|\varphi &
\end{array}$$

As a consequence of Claim V, we know that rewriting at the top level is strongly normalizing. To complete the theorem, we need to show that rewriting at the subexpression level is also strongly normalizing. Let $\mathcal{C}[\_]$ denotes a context; that is, an expression with a "hole." Let $\mathcal{C}[e]$ be the expression obtained by "plugging" e into the hole of $\mathcal{C}[\_]$, provided $\mathcal{C}[e]$ is well formed. Note that plugging an expression into a hole is different from the normal notion of substitution; the former allows free variable to be captured by the context, the latter does not. (See Gunter [11] for more detail on the notion of context.) Then

Claim VI. Suppose $e_1 \rightsquigarrow e_2$ and $\mathcal{C}[e_1]$ is well formed. Then $\mathcal{C}[e_2]$ is well formed and $\|\mathcal{C}[e_1]\|\varphi > \|\mathcal{C}[e_2]\|\varphi$ for all φ.

Proof of Claim VI. Since we have in our possession Claim V, the proof is now a routine induction on the structure of $\mathcal{C}[\_]$. This completes the proof of the theorem. $\square$

Therefore every expression of $\mathcal{RSA}$ of complex object type can be reduced to very simple normal forms. Normal forms can be exploited in many proofs of undefinability by showing that there is no normal form that defines the desired function. Normal forms can also be used to demonstrate results of a different nature. An important example of this sort is the following theorem:

Theorem 3.3 (Conservative extension) *Let $e : s$ be an expression of $\mathcal{RSA}$. Then there is an equivalent expression e' of $\mathcal{RSA}$ such that $ht(e') \leq \max(\{ht(s)\} \cup \{ht(s) \mid s$ is the object type of a free variable in $e\})$.*

Proof. We first rewrite e to a normal form under the rewrite system given earlier. Note that in this normal form of e, any occurrence of $empty(\cdot)$ must appear in a context of the form $empty(e_1 \cup \ldots \cup e_n)$. If each e_i has the form $\{\}$, then we replace $empty(e_1 \cup \ldots \cup e_n)$ by *true*. If some e_i has the form $\{\cdot\}$, then we replace $empty(e_1 \cup \ldots \cup e_n)$ by *false*. If some e_i has the form *if* A *then* B *else* C, then we replace $empty(e_1 \cup \ldots \cup e_n)$ by *if* A *then* $empty(e''_1 \cup \ldots \cup e''_n)$ *else* $empty(e'''_1 \cup \ldots \cup e'''_n)$, where e''_j is e_j if $j \neq i$ and is B if $j = i$, and e'''_j is e_j if $j \neq i$ and is C if $j = i$. This rewrite process clearly terminates.

Note that if every free variable of e has height 0, then the final expression would contain no $empty(\cdot)$. However, if some free variable of e has height greater than 0, then each e_i in each $empty(e_1 \cup \ldots \cup e_n)$ of the final expression must have the forms $\{\}$ or $\pi \ldots \pi\ x$, where x is a free variable of e. It should also be remarked that if all the free variables in the original expression e have height greater than 0, the above additional rewrite steps can be skipped.

Let e' be the final result of the above rewrite process. We verify its height by structural induction on it. Let k be the maximum height of the free variables in e', which is no more than that of e.

Case $e' : s$ is x, $\{\}$, *true*, *false*, c, or $()$. Immediate.

Case $e' : bool$ is $empty\ e''$. Immediate by the discussion above.

Case $e' : \{t\}$ is $\{e''\}$. By hypothesis, $ht(e'') \leq \max(ht(t), k)$. Then $ht(e') = \max(ht(s), ht(e'')) \leq \max(ht(s), k)$.

Case $e' : bool$ is $e_1 =^b e_2$. By hypothesis, $ht(e_1) \leq k$ and $ht(e_2) \leq k$. Then $ht(e') = \max(ht(s), ht(e_1), ht(e_2)) \leq \max(k, ht(s))$.

Case $e' : t_1 \times t_2$ is (e_1, e_2). By hypothesis, $ht(e_1) \leq \max(k, ht(t_1))$ and $ht(e_2) \leq \max(k, ht(t_2))$. Then $ht(e') = \max(ht(s), ht(e_1), ht(e_2)) \leq \max(k, ht(s))$.

Case $e' : s$ is $\pi_1\ e''$ or $\pi_2\ e''$. Then e'' must be a free variable or is a chain of projections on a free variable. The case thus holds.

Case $e' : s$ is *if* e_1 *then* e_2 *else* e_3. By hypothesis, $ht(e_3) \leq \max(k, ht(s))$ and $ht(e_2) \leq \max(k, ht(s))$. Also, by hypothesis, $ht(e_1) \leq \max(k, ht(bool)) \leq \max(k, ht(s))$. Thus, $ht(e') \leq \max(k, ht(s))$.

Case $e' : \{t\}$ is $e_1 \cup e_2$. By hypothesis, $ht(e_1) \leq \max(k, ht(s))$ and $ht(e_2) \leq \max(k, ht(s))$. Then $ht(e') \leq \max(k, ht(s))$.

Case $e' : \{t\}$ is $\{e'' \mid x_1 \in e_1, \ldots, x_n \in e_n\}$. By hypothesis, $ht(e_i) \leq \max(k, 1 + ht(x_1), \ldots, 1 + ht(x_{i-1}))$. Now we show by induction on i that the $1 + ht(x_j)$ can be replaced by 1. Starting with e_1. If e_1 is of the form $empty(\cdot)$, then $ht(x_1) = 0$. Otherwise, e_1 must be a chain of projections on a free variable, then $ht(x_1) < k$. In either case, $ht(e_i) \leq \max(k, 1, 1 + ht(x_2), \ldots, 1 + ht(x_{i-1}))$. The analysis can be repeated for the remaining e_i. Then $ht(e_i) \leq \max(k, 1)$. By hypothesis, $ht(e'') \leq \max(k, ht(t))$. Then $ht(e') = \max(k, ht(s), ht(e''), ht(e_1), \ldots, ht(e_n)) \leq \max(k, ht(s))$. $\square$

Consequently, $\mathcal{NRL}_{i,o,k+1} = \mathcal{NRL}_{i,o,k}$ for all i, o, and $k \geq \max(i, o)$. As remarked earlier, the above theorem implies that the height of input/output dictates the kind of functions that our languages can express. In particular, using intermediate expressions of greater heights does not add expressive power. This is in contrast to languages considered by Abiteboul, Beeri, Grumbach, Gyssens, Hull, Su, Van Gucht, and Vianu [2, 1, 13, 10] where the kind of functions that can be expressed is not characterized by the height of input/output and is sensitive to the height of intermediate operators. The principal difference between our languages and these languages is that *powerset* is not expressible in our languages [7] but is expressible in those other languages. This indicates a non-trivial contribution to expressive power by an operation such as a *powerset*.

This result has a practical significance. Some databases are designed to support nested sets up to a fixed depth of nesting. For example, Jaeschke and Schek [15] consider non-first-normal-form relations in which attribute domains are limited to powersets of simple domains (that is, databases whose height is at most 2). "$\mathcal{NRL}$ restricted to expressions of height 2" is a natural query language for such a database. But knowing that $\mathcal{NRL}$ is conservative at all set heights, one can instead provide the user with the entire language $\mathcal{NRL}$ as a more convenient query language for this database, so long as queries have input/output height not exceeding 2.

Furthermore, expressions having height 1 is syntactically very similar to the flat relational calculus. It is therefore not difficult to show further that every function, from a tuple of flat relations to a flat relation, that is definable in $\mathcal{NRL}$ is also expressible in the flat relational algebra. This is the result first proved by Paredaens and Van Gucht [25] in the context of the nested relational algebra of Thomas and Fischer [27]. The Thomas and Fischer algebra is very restrictive and its operators can be applied only to the topmost level of nested relations. Nevertheless, it is possible to show [36] that the addition of a constant function $\lambda x.\{\{\}\}$ to the Thomas and Fischer algebra yields a query language that is equal in expressive power to $\mathcal{NRL}$. The key to the proof of the conservative extension theorem is the use of normal form. The heart of Paredaens and Van Gucht's proof is also a kind of normal form result. However, the following main distinctions can be made between our results:

- The Paredaens and Van Gucht result is a conservative property with respect to flat relational algebra. This implies $\mathcal{NRL}_{i,o,k+1} = \mathcal{NRL}_{i,o,k}$ for $i = o = 1$. We have generalized this to any i and o.
- The normal form used by Paredaens and Van Gucht is a normal form of logic formulae and the intuition behind their proof is mainly that of logical equivalence and quantifier elimination. In our case, the inspiration comes from a well-known optimization strategy (see Wadler's early paper [32, 33] on this subject). In plain terms, we have evaluated the query without looking at the input and managed to flatten the query sufficiently until all intermediate operators of higher heights are "optimized out." This idea is summarized by the pipeline rule $\{e \mid \Delta_1, x \in \{e' \mid \Delta'\}, \Delta_2\} \leadsto \{e[e'/x] \mid \Delta_1, \Delta', \Delta_2[e'/x]\}$ which eliminates the intermediate set built by $\{e' \mid \Delta'\}$.
- It is clear that $\mathcal{NRL}$ can be given a bag semantics by interpreting $\{\}$ as the empty bag, $\cup$ as bag union, $\{e\}$ as singleton bag, and $\{e \mid \Delta\}$ as bag comprehension. Then $\mathcal{NRL}$ can be used as a nested bag query language. The rewrite rules given earlier are also valid under the bag semantics. Hence the normal form and the proof of the conservative extension theorem above hold for bags as well. This is most useful. For example, it follows easily that the nested bag language obtained by adding a duplicate elimination primitive and the nested bag language obtained by adding a "bag subtraction" primitive define two distinct classes of functions, neither of which is properly included in the other [18]. It is not clear that the proof given by Paredaens and Van Gucht is applicable in this case.
- The setting for this result is also worth mentioning. $\mathcal{NRL}$ can actually be parameterized by an unspecified signature and we do not use any notion of active domain. So extra primitives can be added to the language without affecting strong normalization. Conservative extension is sensitive to new primitives. Nevertheless, $\mathcal{NRL}(p)_{i,o,k+1} = \mathcal{NRL}(p)_{i,o,k}$ continues to hold so long as $k \geq ht(p)$, where p is the extra primitive. For example, we can add $intpowerset : \{int\} \to \{\{int\}\}$ which computes the powerset of a set of integers to $\mathcal{NRL}$. Then any function having input/output of height at most 2 definable in $\mathcal{NRL}(intpowerset)$ can be defined without using intermediate data beyond height 2 [19].

As pointed out, Paredaens and Van Gucht's result involved a certain amount of quantifier elimination. There are several other general results in logic that were proved using quantifier elimination; see Gaifman [9] and Enderton [8]. The pipeline rule is related to quantifier elimination. It corresponds to eliminating quantifier in set theory as $\{e \mid \Delta_1 \wedge (\exists x. x \in \{e' \mid \Delta'\}) \wedge \Delta_2\} \leadsto$

$\{e[e'/x] \mid \Delta_1 \wedge \Delta' \wedge \Delta_2[e'/x]\}$. It is interesting to observe that the logical notion of quantifier elimination corresponds to the physical notion of getting rid of intermediate data. Nevertheless, we stress again that the pipeline rule makes sense across sets and bags (and in the more general form to be given in the next section, across lists as well) but quantifier elimination does not.

4 Extensions to the Main Theorem

In this section, we extend $\mathcal{NRC}$ to $\mathcal{NRC}+$ by a variant type mechanism. Then we provide a proof that this extended language is conservative with respect to set height. Furthermore, the proof holds uniformly when the language is interpreted under a set-, list-, or bag-based semantics.

Types. Variant types are added to the language. If s and t are object types, then the variant type $s+t$ is also an object type. The domain of a variant type $s+t$ is the union of the domains of s and t but values from s are tagged with a 1-tag and values from t are tagged with a 2-tag.

Expressions. Three new constructs are required to manipulate variant objects. Their formation rules are listed in Figure 3 below.

$$\frac{e : s}{\mathit{left}^t\ e : s+t} \qquad \frac{e : s}{\mathit{right}^t\ e : t+s}$$

$$\frac{e_1 : s_1 + s_2 \qquad e_2 : t \qquad e_3 : t}{\mathit{case}\ e_1\ \mathit{of}\ \mathit{left}\ x^{s_1} \ \Rightarrow\ e_2 \mid \mathit{right}\ x^{s_2} \ \Rightarrow\ e_3 : t}$$

Figure 3: Syntax for variants.

Semantics. *left* e injects e into a variant object by tagging the object denoted by e with a 1-tag. *right* e injects e into a variant object by tagging the object denoted by e with a 2-tag. $\mathit{case}\ e_1\ \mathit{of}\ \mathit{left}\ x \ \Rightarrow\ e_2 \mid \mathit{right}\ y \ \Rightarrow\ e_3$ processes the variant object denoted by e_1 as follows. If e_1 is equal to *left* e, then the case expression is equal to $e_2[e/x]$. If e_1 is equal to *right* e, then the case expression is equal to $e_3[e/x]$. That is, the left or the right branch is taken depending on whether e_1 has a 1-tag or a 2-tag respectively.

Examples. Let X denote a set having type $\{s+t\}$. Then $\bigcup\{(\mathit{case}\ x\ \mathit{of}\ \mathit{left}\ y\ \Rightarrow\ \{y\} \mid \mathit{right}\ z\ \Rightarrow\ \{\}) \mid x \in X\}$ has type $\{s\}$ and denotes the selection of items that are 1-tagged in the set X. Variants are really a rational generalization of null values. For example, if an object is either an integer or is null, it can be given the type $\mathit{unit} + \mathit{int}$ and is represented as $\mathit{left}()$ if it is null or as *right* 5 if it is the integer 5.

In the presence of variants, we can identify the Boolean type with the variant type *unit* + *unit*. That is, we treat *true* as a shorthand for *left*(), *false* as a shorthand for *right*(), and *if* e_1 *then* e_2 *else* e_3 as a shorthand for *case* e_1 *of left* $x \Rightarrow e_2 \mid$ *right* $x \Rightarrow e_3$. This identification of *bool* as *unit* + *unit* is used below to give a proof that is simpler to than the proof in the previous section and yet bears a close relationship to it.

Theorem 4.1 *Let* $e : s$ *be an expression of* $\mathcal{NRC}+$. *Then there is an equivalent expression* e' *of* $\mathcal{NRC}+$ *such that* $ht(e') \leq \max(\{ht(s)\} \cup \{ht(s) \mid s$ *is the object type of a free variable in* $e\})$.

Proof. We use the strategy of the previous section and consider the new rewrite system below. The rules of this system corresponds to the rules of the previous system in a direct way. (The same numbering is used.)

1. $(\lambda x.e)e' \leadsto e[e'/x]$
2. $\pi_i(e_1, e_2) \leadsto e_i$
3. $\mathit{case}\ \mathit{left}\ e\ \mathit{of}\ \mathit{left}\ x \Rightarrow e_2 \mid \mathit{right}\ y \Rightarrow e_3 \leadsto e_2[e/x]$
4. $\mathit{case}\ \mathit{right}\ e\ \mathit{of}\ \mathit{left}\ x \Rightarrow e_2 \mid \mathit{right}\ y \Rightarrow e_3 \leadsto e_3[e/y]$
5. $\mathit{case}\ (\mathit{case}\ e_1'\ \mathit{of}\ \mathit{left}\ x' \Rightarrow e_2' \mid \mathit{right}\ y' \Rightarrow e_3')\ \mathit{of}\ \mathit{left}\ x \Rightarrow e_2 \mid \mathit{right}\ y \Rightarrow e_3 \leadsto \mathit{case}\ e_1'\ \mathit{of}\ \mathit{left}\ x' \Rightarrow (\mathit{case}\ e_2'\ \mathit{of}\ \mathit{left}\ x \Rightarrow e_2 \mid \mathit{right}\ y \Rightarrow e_3) \mid \mathit{right}\ y' \Rightarrow (\mathit{case}\ e_3'\ \mathit{of}\ \mathit{left}\ x \Rightarrow e_2 \mid \mathit{right}\ y \Rightarrow e_3)$
6. $\pi_i\ (\mathit{case}\ e_1\ \mathit{of}\ \mathit{left}\ x \Rightarrow e_2 \mid \mathit{right}\ y \Rightarrow e_3) \leadsto \mathit{case}\ e_1\ \mathit{of}\ \mathit{left}\ x \Rightarrow \pi_i\ e_2 \mid \mathit{right}\ y \Rightarrow \pi_i\ e_3$
7. $\bigcup\{e \mid x \in \{\}\} \leadsto \{\}$
8. $\bigcup\{e \mid x \in \{e'\}\} \leadsto e[e'/x]$
9. $\bigcup\{e \mid x \in (e_1 \cup e_2)\} \leadsto (\bigcup\{e \mid x \in e_1\}) \cup (\bigcup\{e \mid x \in e_2\})$
10. $\bigcup\{e_1 \mid x_1 \in \bigcup\{e_2 \mid x_2 \in e_3\}\} \leadsto \bigcup\{\bigcup\{e_1 \mid x_1 \in e_2\} \mid x_2 \in e_3\}$
11. $\bigcup\{e_1 \mid x_1 \in (\mathit{case}\ e_2\ \mathit{of}\ \mathit{left}\ x_2 \Rightarrow e_3 \mid \mathit{right}\ x_3 \Rightarrow e_4)\} \leadsto \mathit{case}\ e_2\ \mathit{of}\ \mathit{left}\ x_2 \Rightarrow \bigcup\{e_1 \mid x_1 \in e_3\} \mid \mathit{right}\ x_3 \Rightarrow \bigcup\{e_1 \mid x_1 \in e_4\}$

It is easy to see that these rewrite rules are sound. That is, if $e_1 \leadsto e_2$, then e_1 and e_2 denote the same value.

Now let $k = \max\{ht(t) \mid t$ is the object type of a free variable in $e\}$. Suppose e has a normal form e' under the above rewrite rules (and the rewrite steps involving *empty* described in the proof of Theorem 3.3). We show by structural induction on e' that e' satisfies the requirement of the theorem. The three more interesting cases are given below.

Case $e' : s$ is $\bigcup\{e_1 \mid x \in e_2\}$ where $e_2 : \{s_2\}$. By hypothesis, $ht(e_2) \leq \max(k, 1)$. So $ht(x) = ht(e_2) - 1 \leq k$. Then, by hypothesis, $ht(e_1) \leq \max(k, ht(x), ht(s)) = \max(k, ht(s))$. Then $ht(e') = \max(ht(s), ht(e_1), ht(e_2)) \leq \max(k, ht(s))$.

Case $e' : s$ is $\mathit{left}^{s_2}\ e_1$ where $e_1 : s_1$. Then s is $s_1 + s_2$. By hypothesis, $ht(e_1) \leq \max(k, ht(s_1))$. So $ht(e') = \max(ht(e_1), ht(s)) \leq \max(k, ht(s))$. The case where $e' : s$ is $\mathit{right}\ e_1$ is similar.

Case $e' : s$ is $\mathit{case}\, e_1\ \mathit{of}\ \mathit{left}\, x \Rightarrow e_2 \mid \mathit{right}\, y \Rightarrow e_3$, where $e_1 : s_1 + s_2$. Then $x : s_1$, $y : s_2$, $e_2 : s$, and $e_3 : s$. By hypothesis, $ht(e_1) \leq k$. Consequently, $ht(s_1) \leq k$ and $ht(s_2) \leq k$. By hypothesis, $ht(e_2) \leq \max(k, ht(x), ht(s)) = \max(k, ht(s))$. Similarly, $ht(e_3) \leq \max(k, ht(y), ht(s)) = \max(k, ht(s))$. Now $ht(e') = \max(ht(e_1), ht(e_2), ht(e_3)) \leq \max(k, ht(s))$.

Finally, we have to show that the normal form e' of e exists. To do this, we prove that the rewrite system is strongly normalizing. Let φ maps variable names to natural numbers greater than 1. Let $\varphi[n/x]$ be the function that maps x to n and agrees with φ on other variables. Let $\|e\|\varphi$, defined below, measure the size of e in the environment φ where each free variable x in e is given the size $\varphi(x)$.

- $\|x\|\varphi = \varphi(x)$
- $\|c\|\varphi = \|()\|\varphi = \|\{\}\|\varphi = 2$
- $\|\pi_1\ e\|\varphi = \|\pi_2\ e\|\varphi = \|\mathit{empty}\ e\|\varphi = \|\mathit{left}\ e\|\varphi = \|\mathit{right}\ e\|\varphi = \|\{e\}\|\varphi = 2 \cdot \|e\|\varphi$
- $\|\lambda x.e\|\varphi = \|e\|\varphi[2/x]$
- $\|(\lambda x.e)(e')\|\varphi = \|e\|\varphi[\|e'\|\varphi/x] \cdot \|e'\|\varphi$
- $\|e_1\ \cup\ e_2\|\varphi = \|(e_1, e_2)\|\varphi = \|e_1\ =^b\ e_2\|\varphi = 1 + \|e_1\|\varphi + \|e_2\|\varphi$
- $\|\bigcup\{e' \mid x \in e\}\|\varphi = (\|e'\|\varphi[\|e\|\varphi/x] + 1) \cdot \|e\|\varphi$
- $\|\mathit{case}\ e_1\ \mathit{of}\ \mathit{left}\ x \Rightarrow e_2\ \mathit{of}\ \mathit{right}\ y \Rightarrow e_3\|\varphi = \|e_1\|\varphi \cdot (1 + \|e_2\|\varphi[\|e_1\|\varphi/x] + \|e_3\|\varphi[\|e_1\|\varphi/y])$

Using arguments similar to (and actually simpler than) that of Theorem 3.2, it is readily verified that whenever $e \leadsto e'$, we have $\|e\|\varphi > \|e'\|\varphi$ for any choice of φ. For example, modulo a few simplifications in notations, the left-hand-side of Rule 5 has measure $\|e'_1\|\varphi + \|e'_1\|\varphi \cdot \|e'_2\|\varphi + \|e'_1\|\varphi \cdot \|e'_3\|\varphi + \|e'_1\|\varphi \cdot \|e_2\|\varphi + \|e'_1\|\varphi \cdot \|e'_2\|\varphi \cdot \|e_2\|\varphi + \|e'_1\|\varphi \cdot \|e'_3\|\varphi \cdot \|e_2\|\varphi + \|e'_1\|\varphi \cdot \|e_3\|\varphi + \|e'_1\|\varphi \cdot \|e'_2\|\varphi \cdot \|e_3\|\varphi + \|e'_1\|\varphi \cdot \|e'_3\|\varphi \cdot \|e_3\|\varphi$. On the other hand, the right-hand-side has size $\|e'_1\|\varphi + \|e'_1\|\varphi \cdot \|e'_2\|\varphi + \|e'_1\|\varphi \cdot \|e'_3\|\varphi + \|e'_1\|\varphi \cdot \|e'_2\|\varphi \cdot \|e_2\|\varphi + \|e'_1\|\varphi \cdot \|e'_3\|\varphi \cdot \|e_2\|\varphi + \|e'_1\|\varphi \cdot \|e'_2\|\varphi \cdot \|e_3\|\varphi + \|e'_1\|\varphi \cdot \|e'_3\|\varphi \cdot \|e_3\|\varphi$. The latter is obviously less than the former. Therefore, the rewrite system is strongly normalizing. This completes the proof. $\square$

As remarked earlier, variant mechanisms have been used in some data models such as Abiteboul and Hull [3] and Hull and Yap [14]. However, many earlier interesting works on expressive power do not consider them [13, 10, 2]. We hope the above result have rectified this situation to some extent.

Our languages have been given semantics based on sets. These languages can be given semantics based on bags or on lists. For example, $\mathcal{NRC}$ can be treated as a "nested bag calculus" by interpreting $\{\}$ as the empty bag, $e_1 \cup e_2$ as union of bags, and $\bigcup\{e' \mid x \in e\}$ as flatmapping the function $\lambda x.e'$ over the bag e. Similarly, $\mathcal{NRC}$ can be treated as a "nested list calculus" by treating $\{\}$ as the empty list, $e_1 \cup e_2$ as the concatenation of list e_1 to the list e_2, and $\bigcup\{e' \mid x \in e\}$ as flatmapping the function $\lambda x.e'$ over the list e. It is easy to check that the rewrite rules given in this section are valid for bag semantics as well as for list semantics. So the same proof above works for "nested bag calculus" and for "nested list calculus." In fact, it works even in the presence of variant types.

The proof is really over the syntax of $\mathcal{NRL}$. It does not matter what semantics is being given to $\mathcal{NRL}$, as long as the equations used in the rewrite rules are sound with respect to that semantics. An important point to note is that the equality test primitive available in the syntax of $\mathcal{NRL}$ is for base types only. Hence the bag and list calculi as mentioned above can perform equality tests at base types only. Equality tests at other types are not necessarily definable in these calculi. However, it is known that these calculi can be enriched with more primitives, including equality tests at all complex object types, and still retain the conservative extension property; see Libkin and Wong [17, 18, 19].

The uniformity of this proof allows us to draw a few useful conclusions. Observe that the translations between $\mathcal{RSA}$ and $\mathcal{NRC}$ preserve set height. Therefore, the conservative extension theorem holds also for "relative bag abstraction" and for "relative list abstraction." It must be remarked that these conclusions cannot be reached from the proof given in Section 3. The proof in Section 3 does not work when $\mathcal{RSA}$ is interpreted using a list semantics. This is because two of the rules used in Section 3 (namely Rules 9 and 11) are not valid as list concatenation does not commute.

In addition, the new proof based on $\mathcal{NRC}$ is also considerably simpler than the proof based $\mathcal{RSA}$ in several ways. Firstly, the rewrite rules for $\mathcal{NRC}$ are clearly simpler than those for $\mathcal{RSA}$. For example, Rule 11 for $\mathcal{NRC}$ has no side condition but Rule 11 for $\mathcal{RSA}$ has side conditions. More significantly, Rules 7 to 11 for $\mathcal{NRC}$ are all "definite" in nature; in contrast, Rules 7 to 11 for $\mathcal{RSA}$ all involve Δ's, which are "indefinite" sequences of $x_i \in e_i$. In other words, implementing the $\mathcal{RSA}$ rules in a real life rewrite system (such as a query optimizer) would be very messy, whereas implementing the $\mathcal{NRC}$ rules would be very straightforward.

Secondly, most of the claims used are proved by structural induction on expressions. Since $\mathcal{RSA}$ and $\mathcal{NRC}$ have the same number of constructs, one would expect the proofs to have similar complexity. However, the proofs involving $\mathcal{RSA}$ are often clumsier than the corresponding ones for $\mathcal{NRC}$. The irregularity of the comprehension construct of $\mathcal{RSA}$ is again the culprit, because when one reaches the case for the $\{e \mid x_1 \in e_1, \ \ldots, \ x_n \in e_n\}$ construct in $\mathcal{RSA}$, one would need to perform a sub-induction on n!

However, $\mathcal{RSA}$ has an important saving grace: Queries and examples written in $\mathcal{RSA}$ are often more readable than the corresponding ones in $\mathcal{NRC}$. Indeed, this readability factor is the reason that we have chosen to present our main result using $\mathcal{RSA}$, even though it would have been considerably more elegant using $\mathcal{NRC}$. Curiously, the comprehension construct of $\mathcal{RSA}$, which is bad from the technical discussion above, is what makes $\mathcal{RSA}$ queries more readable.

5 Conclusion

In summary, we have shown that the conservative property $\mathcal{NRL}_{i,o,k+1} = \mathcal{NRL}_{i,o,k}$ holds at all i, o, and $k \geq \max(i, o)$. Furthermore, we have provided a proof that holds uniformly regardless of whether $\mathcal{NRL}$ is used as a nested relational language, as a nested bag language, or as a nested list language.

It should also be remarked that the same technique can be used to show that the conservative property continues to hold even when rational numbers, rational arithmetics, and a rational summation operator are added to the $\mathcal{NRL}$. The language thus augmented is very interesting because queries such as "select count from column," "select average from column," "select minimum from column," and "select maximum from column" can be expressed. In other words, the language thus endowed with rationals is a conservative extension of SQL. See Libkin and Wong [17]. This property can then be used to prove a powerful finite-cofiniteness result [20], which implies that the language extended with rationals and aggregate functions cannot express recursive queries such as transitive closure.

Also important is the establishment of the strong normalization theorem in Section 3. It induces very simple normal forms for expressions of $\mathcal{RSA}$ under the set and bag semantics. This result can be used to study relative strength of various primitives that one may consider adding to $\mathcal{NRL}$. This direction proves to be fruitful and we have obtained further results on programming with nested bags. See Libkin and Wong [18].

The rewrite rules given in Theorem 4.1 are actually a subset of the rules used in an optimizer for an implementation of $\mathcal{NRL}$. The entire system of rewrite rules retains the strong normalization property. We have also been successful in demonstrating the effectiveness of these rules with respect to a call-by-value evaluation strategy. These rules generalize many well-known algebraic relational optimization identities. For example, Rule 11 of Theorem 4.1 together with another rule of our optimizer — $\bigcup\{(\textit{case } e_1 \textit{ of left } x \Rightarrow e_2 \mid \textit{right } y \Rightarrow e_3) \mid z \in e\} \leadsto (\textit{case } e_1 \textit{ of left } x \Rightarrow \bigcup\{e_2 \mid z \in e\} \mid \textit{right } y \Rightarrow \bigcup\{e_3 \mid z \in e\})$, if $z \notin FV(e_1)$ — is a generalization of the folk wisdom of migrating "filters" towards "generators." See Wong [36].

Acknowledgements. The author thanks Val Tannen and Peter Buneman for many useful discussions and invaluable suggestions, Dirk Van Gucht for explaining a fine point of his conservativeness result, and Catriel Beeri for useful suggestions and editorial advice. The author was supported in part by grants NSF IRI-90-04137 and ARO DAALO3-89-C-0031-PRIME. This

work was done when the author was at the University of Pennsylvania.

References

[1] S. Abiteboul, C. Beeri, M. Gyssens, D. Van Gucht, An introduction to the completeness of languages for complex objects and nested relations, *in* "Nested Relations and Complex Objects in Databases," Lecture Notes in Computer Science, Vol. 361, Springer-Verlag, Berlin, 1989.

[2] S. Abiteboul, C. Beeri, On the power of languages for the manipulation of complex objects, *in* "Proceedings of International Workshop on Theory and Applications of Nested Relations and Complex Objects," Darmstadt, 1988.

[3] S. Abiteboul, R. Hull, IFO: A formal semantic database model, *ACM Transactions on Database Systems* **12**, No. 4 (1987), 525–565.

[4] H. Barendregt, "The Lambda Calculus: Its Syntax and Semantics," Studies in Logic and Foundations of Mathematics, Vol. 103, Elsevier, 1984.

[5] V. Breazu-Tannen, P. Buneman, S. Naqvi, Structural recursion as a query language, *in* "Proceedings of 3rd International Workshop on Database Programming Languages," Morgan Kaufmann, 1991.

[6] V. Breazu-Tannen, R. Subrahmanyam, Logical and computational aspects of programming with Sets/Bags/Lists, *in* "Proceedings of 18th International Colloquium on Automata, Languages, and Programming," Lecture Notes in Computer Science, Vol. 510, Springer Verlag, Berlin, 1991.

[7] V. Breazu-Tannen, P. Buneman, L. Wong, Naturally embedded query languages, *in* "Proceedings of 4th International Conference on Database Theory," Lecture Notes in Computer Science, Vol. 646, Springer-Verlag, Berlin, 1992.

[8] H. B. Enderton, "A Mathematical Introduction to Logic," Academic Press, San Diego, 1972.

[9] H. Gaifman, On local and non-local properties. *in* "Proceedings of the Herbrand Symposium, Logic Colloquium '81," North Holland, 1982.

[10] S. Grumbach, V. Vianu, Playing games with objects, *in* "Proceedings of 3rd International Conference on Database Theory," Lecture Notes in Computer Science, Vol. 470, Springer-Verlag, Berlin, 1990.

[11] C. A. Gunter, "Semantics of Programming Languages: Structures and Techniques," Foundations of Computing, MIT Press, Cambridge, Massachusetts, 1992.

[12] M. Gyssens, D. Van Gucht, A comparison between algebraic query languages for flat and nested databases, *Theoretical Computer Science* **87** (1991), 263–286.

[13] R. Hull, J. Su, On the expressive power of database queries with intermediate types, *Journal of Computer and System Sciences* **43** (1991), 219–267.

[14] R. Hull, C. K. Yap, The format model: A theory of database organisation, *Journal of the ACM* **31**, No. 3 (1984), 518–537.

[15] G. Jaeschke, H. J. Schek, Remarks on the algebra of non-first-normal-form relations, *in* "Proceedings 1st ACM SIGACT/SIGMOD Symposium on Principles of Database Systems," Los Angeles, California, 1982.

[16] L. A. Jategaonkar, J. C. Mitchell, ML with extended pattern matching and subtypes, *in* "Proceedings of ACM Conference on LISP and Functional Programming," Snowbird, Utah, 1988.

[17] L. Libkin, L. Wong, Aggregate functions, conservative extension, and linear orders, *in* "Proceedings of 4th International Workshop on Database Programming Languages," Manhattan, New York, 1993.

[18] L. Libkin, L. Wong, Some properties of query languages for bags, *in* "Proceedings of 4th International Workshop on Database Programming Languages," Manhattan, New York, 1993.

[19] L. Libkin, L. Wong, Conservativity of nested relational calculi with internal generic functions, *Information Processing Letters* **49**, No. 6 (1994), 273–280.

[20] L. Libkin and L. Wong, New techniques for studying set languages, bag languages, and aggregate functions. *in* "Proceedings of 13th ACM Symposium on Principles of Database Systems," Minneapolis, Minnesota, 1994.

[21] R. Milner, M. Tofte, R. Harper, "The Definition of Standard ML," MIT Press, 1990.

[22] E. Moggi, Notions of computation and monads, *Information and Computation* **93** (1991), 55–92.

[23] A. Ohori, P. Buneman, V. Breazu-Tannen, Database programming in Machiavelli: A polymorphic language with static type inference, *in* "Proceedings of ACM-SIGMOD International Conference on Management of Data," Portland, Oregon, 1989.

[24] A. Ohori, "A Study Of Semantics, Types, And Languages For Databases And Object Oriented Programming," PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, 1989.

[25] J. Paredaens, D. Van Gucht, Converting nested relational algebra expressions into flat algebra expressions, *ACM Transaction on Database Systems* **17**, No. 1 (1992), 65–93.

[26] D. Remy, Typechecking records and variants in a natural extension of ML, *in* "Proceedings of 16th Symposium on Principles of Programming Languages," Austin, Texas, 1989.

[27] S. J. Thomas, P. C. Fischer, Nested relational structures, *in* "Advances in Computing Research: The Theory of Databases," JAI Press, 1986.

[28] P. W. Trinder, Comprehension: A query notation for DBPLs, *in* "Proceedings of 3rd International Workshop on Database Programming Languages," Morgan Kaufmann, 1991.

[29] P. W. Trinder, P. L. Wadler, List comprehensions and the relational calculus, *in* "Proceedings of 1988 Glasgow Workshop on Functional Programming," Rothesay, Scotland, 1988.

[30] D. Turner, Recursion equations as a programming language, *in* "Functional Programming and its Applications." Cambridge University Press, 1982.

[31] D. Turner, Miranda—a non-strict functional language with polymorphic types, *in* "Proceedings of Conference on Functional Programming Languages and Computer Architecture," Lecture Notes in Computer Science, Vol. 201, Springer-Verlag, Berlin, 1985.

[32] P. Wadler, Listlessness is better than laziness, *in* "Proceedings of ACM Symposium on Lisp and Functional Programming," Austin, Texas, 1984.

[33] P. Wadler, Listlessness is better than laziness II, *in* "Programs as Data Objects," Lecture Notes in Computer Science, Vol. 217, Springer-Verlag, Berlin, 1985.

[34] P. Wadler, Comprehending monads, *Mathematical Structures in Computer Science* **2** (1992), 461–493.

[35] D. A. Watt, P. Trinder, "Towards a theory of bulk types," Fide Technical Report 91/26, Glasgow University, Glasgow G12 8QQ, Scotland, 1991.

[36] L. Wong, "Querying Nested Collections," PhD Thesis, Department of Computer and Information Sciences, University of Pennsylvania, Philadelphia, PA 19104, August 1994.