On Two Forms of Structural Recursion *

Dan Suciu[1] and Limsoon Wong[2]

[1] University of Pennsylvania, Philadelphia. Email: suciu@saul.cis.upenn.edu
[2] Institute of Systems Science, Singapore. Email: limsoon@iss.nus.sg

Abstract. We investigate and compare two forms of recursion on sets for querying nested collections. The first one is called *sri* and it corresponds to sequential processing of data. The second one is called *sru* and it corresponds to data-parallel processing. A uniform first-order translation from *sru* into *sri* was known from previous work. The converse translation is by necessity more difficult and we have obtained three main results concerning it. First, we exhibit a uniform translation of *sri* queries into *sru* queries over the nested relational algebra. We observe that this translation maps PTIME algorithms into exponential-space queries. The second result proves that *any* uniform translation of *sri* queries into *sru* queries over the nested relational algebra must map some PTIME queries into exponential-space ones. In fact, in the presence of certain external functions, we provide a PTIME *sri* query for which any equivalent *sru* query requires exponential space. Thus, as a mechanism for implementing *algorithms* over complex objects, *sru* is strictly less powerful than *sri*. This inefficiency is in contrast to a previous result that uniformly translates efficient *sri* programs into efficient *sru* programs, but over a language with higher-order functions. Our third result proves that, in the absence of external functions, higher-order functions do not add more expressive power to the nested relational algebra with *sri* or *sru*. However, elimination of higher-order functions cannot be done uniformly, because in the presence of certain external functions, more expressive power can be gained from the higher-order functions. These three results suggest that higher-order functions could be useful in query languages.

1 Introduction

Structural recursion is an attractive paradigm for programming with sets. It can be conceived in two distinct ways, corresponding to the two distinct universal properties that the finite set construction enjoys [7]. The first gives rise to a form of structural recursion that Tannen, Buneman, and Wong call *sri* [8]. It is closely related to the *set_reduce* of Stemple and Sheard [21] and the *fold* operator in functional programming languages [5]. Essentially, *sri* defines a function g on a set O by iteration in an order-independent manner over the elements of O.

* An extended abstract of this paper appeared in the *Proceedings of the 5th International Conference on Database Theory*, Prague, January 1995.

Namely, it allows us to define g by:

$$g(\{\}) = e$$
$$g(\{x\} \cup O) = i(x, g(O))$$

Here i is a previously defined function and e is some value. We write $sri(i, e)$ to denote the function g thus defined. In order for g to be well defined, i has to satisfy the following two conditions: $i(x, i(y, a)) = i(y, i(x, a))$ (commutativity), and $i(x, i(x, a)) = i(x, a)$ (idempotence). This form of recursion is related to the primitive recursion schema in the theory of recursive functions [17].

The second form of structural recursion is called *sru* by Tannen, Buneman, and Wong [8], from *structural recursion on the union presentation*. It is related to the *hom* operator of Machiavelli [18] and the *pump* operator of FAD [4]. Essentially, *sru* allows us to define a function h on a set O in a divide-and-conquer manner. To compute h, one has to divide O recursively in an order-independent way into ever smaller subsets until one reaches sets with 0 or 1 elements. More precisely, *sru* allows us to define some function h by:

$$h(\{\}) = e$$
$$h(\{x\}) = f(x)$$
$$h(O_1 \cup O_2) = u(h(O_1), h(O_2))$$

We use $sru(u, f, e)$ to denote the h thus defined. Note that it is not well defined unless the following conditions are satisfied: $u(e, a) = u(a, e) = a$ (identity), $u(a, b) = u(b, a)$ (commutativity), and $u(a, a) = a$ (idempotence).

When *sri* and *sru* are considered as candidates to be incorporated in a database query language, it is necessary to compare the relative expressive power of the two resultant languages. A straightforward and efficient translation of *sru* into *sri* is described in Tannen and Subrahmanyam [7]. This translation is first-order expressible. This implies that any reasonable language augmented with *sri* is at least as powerful as the same language augmented with *sru*. Furthermore, any efficient *algorithm* expressed in the language with *sru* is tranlated into an efficient one in the language with *sri*. Tannen and Subrahmanyam also give a simple translation of *sri* into *sru*. However, this translation uses higher-order functions. This paper investigates alternative translations of *sri* into *sru* over query languages for complex objects without higher-order functions.

We use the nested relational calculus $\mathcal{NRC}$ of Wong [25] as the ambient query language in this investigation. $\mathcal{NRC}$ has the same power as the query languages for nested relations of Thomas and Fischer [24], Schek and Scholl [20], Colby [9], etc. We add the two forms of structural recursion above to $\mathcal{NRC}$. Our first result is a *uniform* translation of queries in $\mathcal{NRC}(sri)$ into queries in $\mathcal{NRC}(sru)$. This translation works in the presence of arbitrary external functions. Furthermore, it works equally well for other collection types, such as bags and lists. However, the translation is expensive. In particular, it maps some PTIME algorithm into one that requires exponential space.

Our second result proves that *any* uniform translation of queries from $\mathcal{NRC}(sri)$ into $\mathcal{NRC}(sru)$ has to be expensive. In particular, any such uniform translation must map some PTIME algorithms into exponential space ones.

"Uniformity" is a desirable condition that makes separate compilation possible. A *uniform* translation has to translate or compile $sri(i, e)$ by independently translating or compiling sri, i, and e. It should be stressed that this result does not compare the relative expressive power of the two languages, which is the same. Rather, this result compares their ability to express efficient *algorithms*. It is in the same spirit as that of Abiteboul and Vianu [3], proving that *parity* cannot be expressed in PTIME by a Generic Machine, and as that of Suciu and Paredaens [23], proving that Abiteboul and Beeri's algebra for complex objects needs exponential space to express transitive closure. We need to associate to each expression in $\mathcal{NRC}(sru)$ and $\mathcal{NRC}(sri)$ a certain *algorithm*. This association is accomplished by formally defining an evaluation strategy for these expressions [23]. Our result depends on the particular evaluation strategy considered, which is the unoptimized "call-by-value" evaluation strategy, and may not hold for more advanced evaluation techniques. A weakness of our result is that we do not have a particular PTIME query in $\mathcal{NRC}(sri)$ for which no equivalent PTIME query in $\mathcal{NRC}(sru)$ exists. However, given certain external functions, we can construct such a query; see Proposition 3. Still our result suggests that $\mathcal{NRC}(sri)$ can implement more efficient algorithms than $\mathcal{NRC}(sru)$.

In the context of database query languages and ordered databases, it is known [22] that first-order logic extended with *divide and conquer recursion*, which is a form of recursion closely related to sru, captures the complexity class NC, while first-order logic extended with sri captures PTIME. Here NC is the class of functions computable by polynomially many processors in polylogarithmic parallel time. NC is known to be included in PTIME, but it is unknown whether they are equal. Thus our result here (Theorem 2) can be interpreted as saying that there is no uniform translation of a certain query language for PTIME into another query language for NC.

Our third result is motivated by the efficient translation of sri into sru using higher-order functions given by Tannen and Subrahmanyam [7]. While practical programming languages with higher-order functions such as ML and Scheme exists, they have received little attention in the context of database query languages. This efficient translation of sri into sru using higher-order functions together with the proven inefficency of any uniform translation of $\mathcal{NRC}(sri)$ into $\mathcal{NRC}(sru)$ suggests that higher-order functions could be useful in query languages. Our third result proves that the expressive power of $\mathcal{NRC}(sru)$ and $\mathcal{NRC}(sri)$ is not changed by adding higher-order functions to them. However, we also show that higher-order functions can increase expressive power given certain additional primitives.

Organization Section 2 introduces $\mathcal{NRC}$, $\mathcal{NRC}(sru)$, and $\mathcal{NRC}(sri)$. Section 3 presents the uniform translation of sri into sru. This translation proves that $\mathcal{NRC}(sru)$ and $\mathcal{NRC}(sri)$ have the same expressive power, even in the presence of external functions. Section 4 proves that any such translation is very costly, in the sense that the translated query has to create large objects. Section 5 considers the influence of intermediate higher-order functions on the expressive power of our languages. We prove that over complex objects the same expressive power is preserved when higher-order functions are added to our languages; but when certain external functions are also added to our languages, the presence of higher-order functions increases expressive power.

2 Languages to be Studied

The calculus $\mathcal{NRC}$.

As our ambient language in comparing the two forms of structural recursion we use the nested relational calculus $\mathcal{NRC}$ of Wong [25]. It is a language having the same expressive power as other tractable formalisms for complex objects, such as Thomas and Fischer's algebra [24], Schek and Scholl's NF^2 [20], Colby's recursive algebra [9], Abiteboul and Beeri's algebra without powerset [2], Paredaens and Van Gucht's nested algebra [19], etc. $\mathcal{NRC}$ is an algebraic language — the name "calculus" should be taken with the same meaning as in "lambda calculus" and not with the meaning in "first-order calculus" [8].

$\mathcal{NRC}$ is a strongly typed language. The complex object types are given by the grammar:

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

Here b ranges over base types such as numbers $\mathbb{N}$ and booleans $\mathbb{B}$. An object of type $s \times t$ is a pair whose first component is an object of type s and whose second component an object of type t. An object of type $\{s\}$ is a finite set whose elements are objects of type s. The only object of type *unit* is the empty tuple ().

$\mathcal{NRC}$ defines two kinds of expressions: *complex objects expressions* and *function expressions*. A complex object expression denotes a complex object. We write $e : t$ to mean that e is a complex object expression and the complex object it denotes has type t. A function expression denotes a function from complex objects to complex objects. We write $e : s \to t$ to mean that e is a function expression with domain s and codomain t. The language is parameterized by a collection Σ of constants c and external functions p. Each $c \in \Sigma$ has some fixed type t_c and each $p \in \Sigma$ has some type $d_p \to c_p$. For each type s, $\mathcal{X}_s$ is an infinite set of *variables* of type s. $\mathcal{NRC}$ is defined by the following rules:

$$\frac{x \in \mathcal{X}_s}{x : s} \quad \frac{p \in \Sigma}{p : d_p \to c_p} \quad \frac{x \in \mathcal{X}_s \quad e : t}{\lambda x.e : s \to t} \quad \frac{e_1 : s \to t \quad e_2 : s}{e_1\ e_2 : t}$$

$$\frac{c \in \Sigma}{c : t_c} \quad \frac{}{() : unit} \quad \frac{e_1 : s \quad e_2 : t}{(e_1, e_2) : s \times t} \quad \frac{e : s \times t}{\pi_1\ e : s} \quad \frac{e : s \times t}{\pi_2\ e : t}$$

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

$$\frac{}{true : \mathbb{B}} \quad \frac{}{false : \mathbb{B}} \quad \frac{e_1 : \mathbb{B} \quad e_2 : s \quad e_3 : s}{if\ e_1\ then\ e_2\ else\ e_3 : s} \quad \frac{e_1 : s \quad e_2 : s}{e_1 = e_2 : \mathbb{B}}$$

The meanings of these expressions are as follows. $\lambda x.e$ is the function f such that $f(x) = e$. $e_1 e_2$ is *function application*, and applies the function e_1 to the complex object e_2. (e_1, e_2) forms a pair whose first component is e_1 and second component e_2. $\pi_1\ e$ retrieves the first component of the pair e. $\pi_2\ e$ returns the second component of the pair e. $\{\}$ is the empty set. $\{e\}$ forms the singleton set containing e. $e_1 \cup e_2$ is the union of sets e_1 and e_2. $\bigcup\{e_1 \mid x \in e_2\}$ is the set denoted by $sru(\cup,\ \lambda x.e_1,\ \{\})(e_2)$, i.e. $e_1[o_1/x] \cup \ldots \cup e_1[o_n/x]$, where $e_2 = \{o_1, \ldots, o_n\}$. A **query** of type $s \to t$ is a closed function expression $e : s \to t$.

It is convenient to use the syntactic sugar $\{e \mid x_1 \in e_1, \cdots, x_n \in e_n\}$ as a shorthand for $\bigcup\{\bigcup\{\cdots\bigcup\{\{e\} \mid x_n \in e_n\}\cdots \mid x_2 \in e_2\} \mid x_1 \in e_1\}$. Also, we use abbreviations like $\lambda(x, y).x \cup y$ for $\lambda z.\pi_1\ z \cup \pi_2\ z$, or $\bigcup\{x \mid (x, y) \in e\}$, instead of $\bigcup\{\pi_1\ z \mid z \in e\}$.

We write $\mathcal{NRC}(\Sigma)$ to emphasize the external functions present in the language, or simply $\mathcal{NRC}$ when there is no external function. We allow a limited amout of polymophism, like in the *unnest* query: $unnest(x) \stackrel{\text{def}}{=} \{(x, z) \mid (x, y) \in r,\ z \in y\}$, whose type is $\{s \times \{t\}\} \to \{s \times t\}$, for *arbitrary* s and t.

The augmented languages

We extend $\mathcal{NRC}(\Sigma)$ with the two constructions $sru(u, f, e)$ and $sri(i, e)$ defined in Section 1. For this, we add the following typing rules:

$$\frac{u : t \times t \to t \quad f : s \to t \quad e : t}{sru(u, f, e) : \{s\} \to t} \qquad \frac{i : s \times t \to t \quad e : t}{sru(i, e) : \{s\} \to t}$$

We write $\mathcal{NRC}(sri)$, $\mathcal{NRC}(sru)$, $\mathcal{NRC}(sri, \Sigma)$, and $\mathcal{NRC}(sru, \Sigma)$ for the extensions of $\mathcal{NRC}$ and $\mathcal{NRC}(\Sigma)$ with sri and sru respectively. In the remainder of this paper, we consider several aspects of the question of $\mathcal{NRC}(sru, \Sigma) = \mathcal{NRC}(sri, \Sigma)$ for various signatures Σ.

3 Equivalence in the Absence of Higher-Order Functions

In the absence of external functions, $\mathcal{NRC}(sri)$ and $\mathcal{NRC}(sru)$ have the same expressive power as that of Abiteboul and Beeri's algebra (with *powerset*) [8], which in turn has the same expressive power as $\mathcal{NRC}(powerset)$. In this section we extend the equivalence of $\mathcal{NRC}(sri)$ and $\mathcal{NRC}(sru)$ to the equivalence of $\mathcal{NRC}(sri, \Sigma)$ and $\mathcal{NRC}(sru, \Sigma)$, where external functions are present in Σ. We do that by constructing a *uniform* translation between the two languages.

Informally, a **uniform translation** from $\mathcal{NRC}(sri, \Sigma)$ to $\mathcal{NRC}(sru,\ \Sigma)$ is a translation having the following property. Let $\mathsf{i} : s \times t \to t$ and $\mathsf{e} : t$ be two additional variables. Then the translation should map the expression $sri(\mathsf{i}, \mathsf{e})$ to some expression E in $\mathcal{NRC}(sru)$ having i and e as free variables such that $sri(\mathsf{i}, \mathsf{e})$ and E are equivalent, for every instantiation of i and e. In particular, a uniform translation extends to a mapping from $\mathcal{NRC}(sri, \Sigma)$ to $\mathcal{NRC}(sru, \Sigma)$ for an arbitrary set Σ of external functions. Similarly we define uniform translations from $\mathcal{NRC}(sru, \Sigma)$ to $\mathcal{NRC}(sri, \Sigma)$.

From a practical point of view, the uniformity of a translation is connected to the possibility of separate compilation and polymorphism. We want to be able separate the translation or compilation of $sri(i, e)$ from the translation or compilation of i and e. Tannen and Subrahmanyam [7] give the following uniform and efficient translations between sru and sri:

> $sru(u, f, e)(O) \stackrel{\text{def}}{=} sri(\lambda(x, y).u(f(x), y),\ e)(O)$.
> $sri(i, e)(O) \stackrel{\text{def}}{=} sru(U, I, id)(O)(e)$, where id is the identity function, I is a function such that $I(x) = \lambda y.i(x, y)$, and U is function composition, that is $U(x, y) = \lambda z.x(y(z))$.

The first definition gives us a uniform translation from $\mathcal{NRC}(sru)$ to $\mathcal{NRC}(sri)$. The second translation, from sri to sru, assumes a language with higher-order functions. In particular $sru(U, I, id)$, I, and U are all higher-order functions. Unlike general purpose functional languages, database query languages usually do not have higher-order functions and thus cannot express the above uniform translation of sri into sru.

We show that a different uniform translation of sri to sru that does not use higher-order functions is possible. In particular, we give a uniform translation from $\mathcal{NRC}(sri, \Sigma)$ to $\mathcal{NRC}(sru, \Sigma)$.

Theorem 1. *There exists a uniform translation of* $\mathcal{NRC}(sri)$ *into* $\mathcal{NRC}(sru)$. *Therefore, for any set* Σ *of external functions we have:*

$$\mathcal{NRC}(sri, \Sigma) = \mathcal{NRC}(sru, \Sigma)\ .$$

The proof is given in Appendix A. The idea is to observe that the computation of $sru(u, f, e)(O')$ is given by a balanced binary tree, whose leaves are labeled with the elements of O'. In order to compute $sri(i, e)(O)$ where $O = \{o_1, \ldots, o_n\}$, one has to apply i consecutively n times. One can do that with a balanced binary tree, provided the tree has height n and therefore 2^n leaves. Thus, in the translation, we define O' to be a set with 2^n elements.

The proof of the theorem above is robust in the sense that it extends directly to languages for bags and lists. See Tannen and Subrahmanyam [7] for a discussion of how sru and sri extend to these collection types. See also Libkin and Wong [16, 26] for extensions of $\mathcal{NRC}$ to languages for bags and lists.

Due to the way our translation works, a PTIME query in $\mathcal{NRC}(sri)$ may get translated into a query in $\mathcal{NRC}(sru)$ that has exponential complexity. For example, the implementation of transitive closure in $\mathcal{NRC}(sri)$ by Tannen, Buneman, and Naqvi [6] has polynomial complexity. But its translation to $\mathcal{NRC}(sru)$ via Theorem 1 results in an implementation having exponential complexity. Of course, this does not rule out the existence of efficient expressions for transitive closure in $\mathcal{NRC}(sru)$. Nonetheless, we conjecture that there is no PTIME expression for transitive closure in $\mathcal{NRC}(sru)$.

4 The Cost of the Translation

We prove in this section that any uniform first-order translation from $\mathcal{NRC}(sri)$ to $\mathcal{NRC}(sru)$ is inherently expensive. More precisely, we show that for a certain external function i and constant e, the expression $sri(\mathsf{i}, \mathsf{e})$ denotes a query for which any "algorithm" in $\mathcal{NRC}(sru)$ requires exponential space, whereas the "algorithm" denoted by $sri(\mathsf{i}, \mathsf{e})$ is in PTIME. This shows a mismatch in the ability of $\mathcal{NRC}(sri, \Sigma)$ and $\mathcal{NRC}(sru, \Sigma)$ in expressing *algorithms*. It also implies that any expression in $\mathcal{NRC}(sri)$ in which sri is used in an essential way, is mapped by any uniform translation into an expression in $\mathcal{NRC}(sru)$ denoting an exponential-space algorithm.

This result is of a different nature than inexpressibility results showing that a database query language cannot express certain queries. It states that of the two equivalent languages, $\mathcal{NRC}(sri, \Sigma)$ and $\mathcal{NRC}(sru, \Sigma)$, the former can express certain queries in a more efficient way than the latter. It is in the same spirit as the result [3] stating that *parity* cannot be expressed in PSPACE on a Generic Machine and the result [23] stating that $\mathcal{NRA}(powerset)$ cannot implement transitive closure efficiently (even though it *can* express it.) Our result here is somewhat weaker than these: in the absence of external functions we do not know any particular query that can be expressed more efficiently in $\mathcal{NRC}(sri)$ than $\mathcal{NRC}(sru)$. However, we conjecture that $\mathcal{NRC}(sru)$ cannot express transitive closure in PTIME, while $\mathcal{NRC}(sri)$ obviously can [6].

We start by defining an evaluation strategy for $\mathcal{NRC}(\Sigma)$. As in Suciu and Paredaens [23], this strategy is essentially the *eager* or *call-by-value* evaluation strategy in programming languages, presented in a natural deduction style. (See Hennessy [13] for an introduction to this style of operational semantics.) Our result holds *only* in conjunction with this evaluation strategy. It may fail for a *lazy* or *call-by-need* strategy. However, it does not depend on the particular form of presentation.

First, let us define a grammar for *complex objects* O:

$$O ::= c \mid \mathit{true} \mid \mathit{false} \mid () \mid (O, O) \mid \{O, \ldots, O\}$$

where c stands for some constant such as a number, a string, etc.; (O_1, O_2) is a pair; and $\{O_1, \ldots, O_n\}$ is a set. The notation for a complex objects is subject to the restriction that the sets have no duplicates and we consider two sets to be equal if they differ only in the order of their elements. For example, $\{c_1, c_2\}$ and $\{c_2, c_1\}$ denote the same complex object, while $\{c_1, c_2, c_1\}$ is illegal. Complex objects are *typed*; for example, $\{O_1, O_2, O_3\}$ is of type $\{t\}$, provided that O_1, O_2, O_3 are all of type t.

Recall that $\mathcal{NRC}$ has two kinds of expressions: complex object expressions and function expressions. We want to define a binary relation $e \Downarrow O$ between an expression e and a complex object O, by induction on the structure of e. Its intended meaning is "e fully evaluates to O." However, since $\mathcal{NRC}$ is a language with variables, we are forced to evaluate expressions e with free variables, which have been bound to complex objects. Following Curien [11] we define an **environment** to be a set of the form $\rho = \{x_1 := O_1, \ldots, x_n := O_n\}$, where x_1, ..., x_n are variables and $O_1, \ldots, O_n$ are complex objects. Finally we define below the relations $\rho \bullet e \Downarrow O$ and $\rho \bullet e(O) \Downarrow O'$, with the intended meaning: "under the environment ρ, the complex object expression e fully evaluates to O" and "under the environment ρ, the function expression e applied to O fully evaluates to O'." We write $\rho, x := O$ instead of $\rho \cup \{x := O\}$.

$$\frac{c \in \Sigma}{\rho \bullet c \Downarrow c} \qquad \frac{p \in \Sigma \quad p(O) = O'}{\rho \bullet p(O) \Downarrow O'} \qquad \frac{}{\{\ldots, x := O, \ldots\} \bullet x \Downarrow O}$$

$$\frac{\rho, x := O \bullet e \Downarrow O'}{\rho \bullet (\lambda x.e)(O) \Downarrow O'} \qquad \frac{\rho \bullet e_2 \Downarrow O \quad \rho \bullet e_1(O) \Downarrow O'}{\rho \bullet e_1\ e_2 \Downarrow O'}$$

$$\frac{}{\rho \bullet () \Downarrow ()} \qquad \frac{\rho \bullet e_1 \Downarrow O_1 \quad \rho \bullet e_2 \Downarrow O_2}{\rho \bullet (e_1, e_2) \Downarrow (O_1, O_2)} \qquad \frac{\rho \bullet e \Downarrow (O_1, O_2)}{\rho \bullet \pi_1 e \Downarrow O_1} \qquad \frac{\rho \bullet e \Downarrow (O_1, O_2)}{\rho \bullet \pi_2 e \Downarrow O_2}$$

$$\frac{}{\rho \bullet \{\} \Downarrow \{\}} \qquad \frac{\rho \bullet e \Downarrow O}{\rho \bullet \{e\} \Downarrow \{O\}} \qquad \frac{\rho \bullet e_1 \Downarrow O_1 \quad \rho \bullet e_2 \Downarrow O_2}{\rho \bullet e_1 \cup e_2 \Downarrow O} \ \text{where } O = O_1 \cup O_2$$

$$\frac{\rho \bullet e_2 \Downarrow \{O_1, \ldots, O_n\} \quad \rho, x := O_1 \bullet e_1 \Downarrow O'_1 \quad \ldots \quad \rho, x := O_n \bullet e_1 \Downarrow O'_n}{\rho \bullet \bigcup\{e_1 \mid x \in e_2\} \Downarrow O'} \ \text{where } O' = O'_1 \cup \ldots \cup O'_n$$

$$\frac{}{\rho \bullet \mathit{true} \Downarrow \mathit{true}} \qquad \frac{}{\rho \bullet \mathit{false} \Downarrow \mathit{false}}$$

$$\frac{\rho \bullet e_1 \Downarrow \mathit{true} \quad \rho \bullet e_2 \Downarrow O}{\rho \bullet (\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \Downarrow O} \qquad \frac{\rho \bullet e_1 \Downarrow \mathit{false} \quad \rho \bullet e_3 \Downarrow O}{\rho \bullet (\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \Downarrow O}$$

Recall that a query is a function expression e with no free variables. The evaluation $\{\} \bullet e(O) \Downarrow O'$ of a query e applied to a complex object O under this operational semantics should be viewed as a *derivation tree* whose root is labeled by $\{\} \bullet e(O) \Downarrow O'$ and whose nodes are labeled by the rules above. Note that for a certain query e, the height of the tree is independent of O but its width may vary with O (see the rule for $\bigcup\{e_1 \mid x \in e_2\}$).

This operational semantics is extended to $\mathcal{NRC}(sru, \Sigma)$ by adding the following three evaluation rules for sru.

$$\frac{\rho \bullet e \Downarrow O}{\rho \bullet sru(u, f, e)(\{\}) \Downarrow O} \qquad \frac{\rho \bullet f(O) \Downarrow O'}{sru(u, f, e)(\{O\}) \Downarrow O'}$$

$$\frac{\begin{array}{c}\rho \bullet sru(u, f, e)(\{O_1, \cdots, O_{\lfloor n/2 \rfloor}\}) \Downarrow O' \\ \rho \bullet sru(u, f, e)(\{O_{\lfloor n/2 \rfloor + 1}, \cdots, O_n\}) \Downarrow O'' \\ \rho \bullet u(O', O'') \Downarrow O \quad n \geq 2\end{array}}{\rho \bullet sru(u, f, e)(\{O_1, \cdots, O_n\}) \Downarrow O}$$

Note that the evaluation of expressions in $\mathcal{NRC}(sru)$ is not deterministic due to the last rule for sru, which says that we may choose any way of splitting the set $\{O_1, \ldots, O_n\}$ into two roughly equal disjoint halves. However, the well-definedness preconditions on sru guarantee the uniqueness of the output. Also, in contrast to $\mathcal{NRC}(\Sigma)$, for queries e in $\mathcal{NRC}(sru)$, the height of the evaluation tree for $\{\} \bullet e(O) \Downarrow$ varies with O. But the height is bounded by a polylogarithm of the size of O.

Similarly, we extend the operational semantics for $\mathcal{NRC}(\Sigma)$ to $\mathcal{NRC}(sri, \Sigma)$ by adding the two rules below. Note that the height of an evaluation tree $\{\} \bullet e(O) \Downarrow$ is bounded by a polynomial of the size of O.

$$\frac{\rho \bullet e \Downarrow O}{\rho \bullet sri(i,e)(\{\}) \Downarrow O}$$

$$\frac{\rho \bullet sri(i,e)(\{O_2,\cdots,O_n\}) \Downarrow O \quad \rho \bullet i(O_1,O) \Downarrow O'}{sri(i,e)(\{O_1,\ldots,O_n\}) \Downarrow O'}$$

Next, we define the **size** of complex objects. Up to a constant factor, $\mathsf{size}(O)$ gives us the number of symbols needed in the notation for O, assuming one symbol suffices for any value at base types. Formally, $\mathsf{size}(c) = 1$, for constants c of base types, $\mathsf{size}(\mathit{true}) = \mathsf{size}(\mathit{false}) = \mathsf{size}(()) = 1$, $\mathsf{size}(O_1, O_2) = \mathsf{size}(O_1) + \mathsf{size}(O_2) + 1$, and $\mathsf{size}\{O_1,\cdots,O_n\} = \mathsf{size}(O_1) + \cdots + \mathsf{size}(O_n) + 1$.

We define $\mathsf{complex}(T) = \max(\{\mathsf{size}(O'') \mid O'' \text{ occurs in } T\})$, for some derivation tree T. For a query e and complex object O, the **complexity** of the evaluation $\{\} \bullet e(O) \Downarrow$ is defined to be: $\mathsf{complex}(e, O) = \min(\{\mathsf{complex}(T) \mid T \text{ is a derivation tree of } \{\} \bullet e(O) \Downarrow O'\})$.

Since the height of a tree T is polynomially related to $\mathsf{complex}(T)$, the latter gives a rough approximation of the space needed to compute $e(O)$, according to the choices embodied by T. Then $\mathsf{complex}(e, O)$ gives us the smallest amount of space needed to evaluate $e(O)$, among all possible nondeterministic choices.

Now suppose there is a measure $\# : b \to \mathbb{N}$ on complex objects of base type b. Extend this measure to all types as follows:

$\#(c) = 0$, if $t_c \neq b$.
$\#(O_1, O_2) = \max(\#(O_1), \#(O_2))$.
$\#\{O_1,\cdots,O_n\} = \max(\#(O_1),\cdots,\#(O_n))$.

This measure is domain dependent and unrelated to the size of the complex objects. It only becomes interesting through its interaction with the external functions in Σ. Throughout this section, we assume that any external function p in Σ only increases the measure of its argument by a constant amount; that is, there is c_p such that for every O, we have $\#(p(O)) \leq c_p + \#(O)$. This means that, for any evaluation tree T of $e(O)$, we have $\#(e(O)) \leq \#(O) + c \cdot \mathit{height}(T)$ for some constant c, depending only on Σ, leading us to the following theorem:

Theorem 2. *Suppose that every primitive p in the language increases the measure of its input object only by a constant amount. Then the following hold:*

1. *For every query e in $\mathcal{NRC}(\Sigma)$ or $\mathcal{NRC}(powerset, \Sigma)$, there is a constant c_e such that for every complex object O, we have $\#(e(O)) \leq c_e + \#(O)$.*
2. *For every query e in $\mathcal{NRC}(sru, \Sigma)$, there are constants c_e and p_e such that for every complex object O, we have $\#(e(O)) \leq c_e \cdot \log^{p_e}(\mathsf{complex}(e, O)) + \#(O)$.*
3. *For every query e in $\mathcal{NRC}(sri, \Sigma)$, there are constants c_e and p_e such that for every complex object O, we have $\#(e(O)) \leq c_e \cdot (\mathsf{complex}(e, O))^{p_e} + \#(O)$.*

The proof is given in Appendix B. The intuition behind it is that the measure of complex objects may increase only by applying external functions, and that the number of successive applications of external functions is bounded by the depth of the derivation tree. Hence it suffices to observe that the depth of the derivation tree T for an expression e is: constant, when e is in $\mathcal{NRC}(\Sigma)$ or $\mathcal{NRC}(powerset, \Sigma)$; polylogarithmic in the complexity of T, when e is in $\mathcal{NRC}(sru, \Sigma)$; and polynomial in the complexity of T, when e is in $\mathcal{NRC}(sri, \Sigma)$. The function *powerset* is handled as an external function, and it preserves the measure of its input.

The bounds given above are also tight. In fact, we prove that in the presence of certain external functions, *sri* is strictly more efficient than *sru*. Indeed, suppose Σ contains two base types $\mathbb{N}$ and $\mathbf{D}$. $\mathbb{N}$ is the type of natural numbers, while $\mathbf{D}$ contains pairs (x, X), with $x \in \mathbb{N}$ and X a finite set of natural numbers. On these types we have an external function $\mathsf{i} : \mathbb{N} \times \mathbf{D} \to \mathbf{D}$ and a constant $\mathsf{e} : \mathbf{D}$, defined by: $\mathsf{i}(x, (y, X)) = \mathit{if}\ x \in X\ \mathit{then}\ (y, X)\ \mathit{else}\ (1 + y, \{x\} \cup X)$, $\mathsf{e} = (0, \{\})$. Note that, from the point of view of $\mathcal{NRC}(\Sigma)$, the values of type $\mathbf{D}$ are *atomic* values as they can only be accessed through i, e, and the equality test. For example, $\pi_1(\mathsf{e})$ is *not* a well-defined expression in $\mathcal{NRC}(\Sigma)$ because it is illegally typed. Consider the query $f : \{\mathbb{N}\} \to \mathbf{D}$ in $\mathcal{NRC}(sri, \Sigma)$, $f \stackrel{\mathrm{def}}{=} sri(\mathsf{i}, \mathsf{e})$, with meaning $f(X) = (card(X), X)$, of polynomial (in fact linear) complexity.

Corollary 3. *Any expression in $\mathcal{NRC}(sru, \Sigma)$ equivalent to f has exponential complexity. Specifically, for any equivalent query g in $\mathcal{NRC}(sru, \Sigma)$, there are constants p and c such that the complexity of $g(X)$ is*

$$\mathsf{complex}(g, X) = \Omega(2^{\sqrt[p]{card(X)/c}}) \ .$$

Proof. Define the measure of an object of type $\mathbf{D}$ to be $\#(n, X) = n$. By Theorem 2, we have $card(X) = \#(g(X)) \leq c_g \log^{p_g}(\mathsf{complex}(g, X)) + 0$ (since $\#(X) = 0$). Hence, $\mathsf{complex}(g, X) \geq 2^{\sqrt[p_g]{card(X)/c_g}}$. □

Consequently, any uniform translation of $\mathcal{NRC}(sri)$ into $\mathcal{NRC}(sru)$ must map the PTIME expression e for transitive closure of Tannen, Buneman, and Naqvi [6] into an expression with exponential complexity. Indeed, e applies *sri* to a set of size n. Due to the uniformity condition, the derivation tree for the translation of e must contain a subtree isomorphic to the derivation tree of $g(O)$ above, for a set O with n elements. Hence it has at least the complexity of $g(O)$.

This inefficiency is in contrast to the higher-order translation of Tannen and Subrahmanyam [7], which is both uniform and efficient (that is, preserves complexity classes). A similar situation was described by Colson [10]. He gave examples of functions that can be efficiently computed using primitive recursion when functional input and output are allowed, but cannot be efficiently computed when functional input and output are not allowed. This result, as well as that of Colson, indicates the important effects higher-order functions have on performance.

We should also remark on the *hom* operator of Machiavelli [18], called *dcr* elsewhere [22]. If $sru(u, f, e)$ is well defined, then so is $hom(u, f, e)$ and they compute the same function. The difference is that $hom(u, f, e)$ does not have to obey idempotence axiom. Instead, it is required to satisfy $hom(u, f, e)(O_1 \cup O_2) =$

$u(hom(u,f,e)(O_1), hom(u,f,e)(O_2))$ when O_1 and O_2 are disjoint sets. Consequently, the translation given in Section 3 can also be used to implement *sri* in terms of *hom*. More interestingly, the proof of Theorem 2 holds also when *hom* is used instead of *sru*. In the view of the result that *sri* and *dcr* correspond to the complexity classes PTIME and NC respectively [22], Theorem 2 can be interpreted as saying that there is no uniform translation of a language for PTIME into a language for NC.

Since any uniform translation from *sri* to *sru* requires exponential space, it cannot be expressed only over flat relations. Let us denote with $\mathcal{NRC}^1$ the restriction of $\mathcal{NRC}$ to types of set height ≤ 1; it is known [6, 25] that $\mathcal{NRC}^1$ has the same expressive power as the relational algebra. Then, using Theorem 2, we can prove that, over the base type $\mathbb{N}$ with constant 0 and external function *succ*, *sri* is strictly more powerful than *sru* over flat relations:

Corollary 4. $\mathcal{NRC}^1(sru, \mathbb{N}, 0, succ) < \mathcal{NRC}^1(sri, \mathbb{N}, 0, succ)$.

The proof is given in Appendix C. This result is connected to a result by Immerman [15] stating that $QR[\log^k n] < FO + LFP$, for any $k \geq 1$. That result does not imply our result, since it is not obvious whether $\mathcal{NRC}^1(sru)$ is included in $\bigcup_k QR[\log^k n]$. However, his result is somewhat stronger than ours, because the presence of 0 and *succ* is not needed.

5 Higher-Order Functions

Consider now an extension of $\mathcal{NRC}(sri, \Sigma)$ and $\mathcal{NRC}(sru, \Sigma)$ to languages with higher-order functions, which we call $\mathcal{HNRC}(sri, \Sigma)$ and $\mathcal{HNRC}(sru, \Sigma)$ respectively. The interesting aspect of these languages is that they are able to use structural recursion in order to compute functions, in the style of Tannen and Subrahmanian's translation of *sri* into *sru* with higher-order functions.

Since computing with sets needs an equality predicate on its elements, we do not allow the construction of sets of function in these higher-order languages. Hence we separate the types into two levels: the complex object types s, t and the functional types S, T:

$$\begin{aligned} s, t &::= b \mid unit \mid s \times t \mid \{s\} \\ S, T &::= s \mid t \mid S \times T \mid S \to T \end{aligned}$$

The operations at the object level are the same those for $\mathcal{NRC}(sru)$ and $\mathcal{NRC}(sri)$ respectivley, but *sru* and *sri* are extended over functions as well:

$$\frac{e : T \quad i : s \times T \to T}{sri(i, e) : \{s\} \to T} \qquad \frac{e : T \quad f : s \to T \quad u : T \times T \to T}{sru(u, f, e) : \{s\} \to T}$$

Also, we have pairing, projections, lambda abstraction, and application at the function level:

$$\frac{e : T}{\lambda x.e : S \to T} \qquad \frac{e_1 : S \to T \quad e_2 : S}{e_1 e_2 : T}$$

$$\frac{e_1 : S \quad e_2 : T}{(e_1, e_2) : S \times T} \qquad \frac{e : S \times T}{\pi_1 e : S} \qquad \frac{e : S \times T}{\pi_2 e : T}$$

In this section we show that every first-order function definable in these higher-order languages is also definable in their first-order fragments. That is, if $f : s \to t$ can be expressed using some expression containing higher-order functions, then we can find another expression having no higher-order function to implement it. The proof requires several technical definitions.

Define the first-order "equivalent" of a type S to be $rel(S)$ as: $rel(unit) \stackrel{\text{def}}{=} unit$, $rel(\mathbb{B}) \stackrel{\text{def}}{=} \mathbb{B}$, $rel(b) \stackrel{\text{def}}{=} b$, $rel(S \times T) \stackrel{\text{def}}{=} rel(S) \times rel(T)$, $rel(\{s\}) \stackrel{\text{def}}{=} \{rel(s)\}$, and $rel(S \to T) \stackrel{\text{def}}{=} \{rel(S) \times rel(T)\}$. Note that if S is a complex object type, then $S = rel(S)$.

Define the "restriction" of the domain of a type S with respect to a finite set b_0 of objects of type b, $restrict_{b_0}(S) \subseteq rel(S)$, by: $restrict_{b_0}(unit) \stackrel{\text{def}}{=} \{()\}$, $restrict_{b_0}(\mathbb{B}) \stackrel{\text{def}}{=} \{true, false\}$, $restrict_{b_0}(b) \stackrel{\text{def}}{=} b_0$, $restrict_{b_0}(S \times T) \stackrel{\text{def}}{=} \{(x, y) \mid x \in restrict_{b_0}(S),\ y \in restrict_{b_0}(T)\}$, $restrict_{b_0}(\{s\}) \stackrel{\text{def}}{=} \{x \mid x \subseteq restrict_{b_0}(s)\}$, and $restrict_{b_0}(S \to T) \stackrel{\text{def}}{=} \{f \mid f \in \{restrict_{b_0}(S) \times restrict_{b_0}(T)\}$, f is functional$\}$. Note that if b_0 is given, then $restrict_{b_0}(S)$ can be expressed using $\mathcal{NRC}(sri)$, and that $restrict_{b_0}(S) : \{rel(S)\}$.

Define $restrict^S_{b_0} : S \to restrict_{b_0}(S)$ to be a partial function "restricting" an object $O : S$ with respect to a finite subset b_0 of objects of type b. Namely: $restrict^{unit}_{b_0}() \stackrel{\text{def}}{=} ()$; $restrict^{\mathbb{B}}_{b_0}(x) \stackrel{\text{def}}{=} x$; $restrict^b_{b_0}(x) \stackrel{\text{def}}{=} x$, if $x \in b_0$; $restrict^{S \times T}_{b_0}(x, y) \stackrel{\text{def}}{=} (restrict^S_{b_0}(x), restrict^T_{b_0}(y))$, if $restrict^S_{b_0}(x)$ and $restrict^T_{b_0}(y)$ are defined; $restrict^{\{s\}}_{b_0}(X) \stackrel{\text{def}}{=} \{restrict^s_{b_0}(x) \mid x \in X\}$, if $restrict^s_{b_0}(x)$ is defined for each $x \in X$; $restrict^{S \to T}_{b_0}(f) \stackrel{\text{def}}{=} \{(x, y) \mid x = restrict^S_{b_0}(x')$ and $y = restrict^T_{b_0}(y')$ where $y' = f(x')\}$, if the latter is functional. In all other cases $restrict^S_{b_0}(O)$ is undefined. Note that if $o \in restrict_{b_0}(s)$, then $restrict^s_{b_0}(o) = o$. For a first-order function $f : s \to t$, $restrict^{s \to t}_{b_0}(f)$ is precisely the graph of f restricted to b_0.

A function $p : s \to t$ is said to be **internal** in the sense of Hull [14] if $restrict^t_{b_0}(p(o)) = p(o)$ for all finite $b_0 \subseteq b$ for which $o = restrict^s_{b_0}(o)$. In other words, a function is internal if it does not generate new values.

Proposition 5. *Let all external primitives mentioned in Σ be internal. Let $e : S$ be an expression of $\mathcal{HNRC}(sri, \Sigma)$ with $x_1 : S_1, \ldots, x_n : S_n$ as free variables. Let b_0 be a finite subset of b that includes all constants $c : b$ occurring in e. Then there is an expression $rel(e) : rel(S)$ of $\mathcal{NRC}(sri, \Sigma)$ with $\bar{x_1} : rel(S_1), \ldots, \bar{x_n} : rel(S_n)$ as free variables such that for all $O_1 : S_1, \ldots, O_n : S_n$, if $restrict^{S_i}_{b_0}(O_i)$ is defined for each i, then $restrict^S_{b_0}(e[O_1/x_1, \ldots, O_n/x_n])$ is defined and equals $rel(e)[restrict^{S_1}_{b_0}(O_1)/\bar{x_1}, \ldots, restrict^{S_n}_{b_0}(O_n)/\bar{x_n}]$.*

The proof is given in Appendix D. The main idea used in the proof is that hereditarily-finite functions can be represented by hereditarily-finite sets, and hence we can simulate any computation at higher order with some computation over complex objects. So for any signature satisfying the same conditions, we conclude that using higher-order functions at intermediate stages has no influence on the class of first-order functions expressible in our languages.

Theorem 6. *Let all external primitives mentioned in Σ be internal. Suppose $f : s \to t$ is a first-order function in $\mathcal{HNRC}(sri, \Sigma)$, or $\mathcal{HNRC}(sru, \Sigma)$. Then f is also expressible in $\mathcal{NRC}(sri, \Sigma)$, or $\mathcal{NRC}(sru, \Sigma)$ respectively.*

Proof. We prove the case for $\mathcal{HNRC}(sri, \Sigma)$ and $\mathcal{NRC}(sri, \Sigma)$; the other case is similar. Suppose $f : s \to t$ is defined by $\lambda x.e$. Let b_0 be treated as a new constant of type $\{b\}$. Apply the proposition above to e to get $rel(e)$ with $\bar{x}$ as its only free variable and with b_0 appearing as a constant. Let $C : \{b\}$ be the collection of constants $c : b$ appearing in e. Let $D^s : s \to \{b\}$ be the function that extracts objects of type b from its input. This function is readily definable in $\mathcal{NRC}$. Since s and t are complex object types, we know that $s = rel(s)$ and $t = rel(t)$. Then for any $o : s$, we have $f(o) = rel(e)[o/\bar{x}, (C \cup D^s(o))/b_0]$. So $\lambda\bar{x}.rel(e)[(C \cup D^s(\bar{x}))/b_0]$ is an expression in $\mathcal{NRC}(sri, \Sigma)$ implementing f. $\square$

However, in the presence of certain external functions that are not internal, higher-order functions give more expressive power. Indeed, consider the signature Σ containing the base type $\mathbb{N}$ and the external functions $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, $\dot{-} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, and $gen : \mathbb{N} \to \{\mathbb{N}\}$. $+$ is the usual addition operation. $\dot{-}$ is the one-sided subtraction operation; that is, $m \dot{-} n = m - n$ if $m > n$ and $m \dot{-} n = 0$ otherwise. gen is the function $gen(n) = \{0, \ldots, n\}$. Then we have

Theorem 7. *1. The class of functions $f : \mathbb{N} \times \ldots \times \mathbb{N} \to \mathbb{N}$ representable in $\mathcal{NRC}(sri, \Sigma)$ coincides with the class of primitive recursive functions [16].*
2. All functions $f : \mathbb{N} \times \ldots \times \mathbb{N} \to \mathbb{N}$ expressible in Godel's system T [12] are also expressible in $\mathcal{HNRC}(sri, \Sigma)$. Hence $\mathcal{HNRC}(sri, \Sigma)$ can express the Ackermann function [12].

Hence, for this particular Σ, $\mathcal{HNRC}(sri, \Sigma)$ is strictly more expressive than $\mathcal{NRC}(sri, \Sigma)$. More, Theorem 1 also implies that $\mathcal{HNRC}(sru, \Sigma)$ is strictly more expressive than $\mathcal{NRC}(sru, \Sigma)$.

6 Conclusion and Future Work

In summary, we presented three results in this paper. First, we exhibited a uniform first-order translation of *sri* into *sru*. Second, we proved that any such translation must create very large intermediate data and hence caused a degradation in performance. Third, we proved that every first-order function definable using higher-order structural recursion is also definable using first-order structural recursion. In the course of this work, we have observed several interesting phenomena that we would like to pursue further:

As concluded in Section 4, the ability to define higher-order functions leads to considerably more efficient programs. However, such functions are often excluded from query languages as can be seen from the survey of Abiteboul and Kanellakis [1]. As performance is an important factor in database queries, we should begin to pay serious attention to the use of higher-order functions in query languages.

As remarked in Section 4, *sru* can be replaced by *hom* everywhere. Also, there exists a uniform translation from *hom* to *sru*, namely going through *sri*, but which is inefficient. Does a result similar to Theorem 2 hold between *sru* and *hom*, stating that *any* uniform translation from *hom* to *sru* is inefficient?

In our search for the proof in Section 3, we found that the greatest difficulty was in discovering a u which is both idempotent and associative. For example, let $A \circ B$ be composition of binary relations, and consider the following expression for transitive closure: $tc(R) \stackrel{\text{def}}{=} hom(u, f, e)(R)$, where $e \stackrel{\text{def}}{=} \{\}$, $f(x) \stackrel{\text{def}}{=} R$, and $u(A, B) \stackrel{\text{def}}{=} A \cup B \cup ((A \cup B) \circ (A \cup B))$. Replacing *hom* with *sru* will not give us a correct expression, because u is not idempotent. In fact we do not know how to define an efficient transitive closure function using *sru*. We conjecture that there is no efficient expression for transitive closure in $\mathcal{NRC}(sru)$.

Acknowledgements We thank Peter Buneman and Val Tannen, they provided the inspiration for this paper, as well as some useful simplification. We thank Leonid Libkin for discussions. Suciu is supported by National Science Foundation grants CCR-90-57570 and IRI-90-04137. Wong is supported by Army Reseach Office grant DAALO3-89-C-0031-PRIME. This research was carried out when Wong was at the University of Pennsylvania.

References

1. S. Abiteboul and P. Kanellakis. Database theory column: Query languages for complex object databases. *SIGACT News*, 21(3):9–18, 1990.
2. S. Abiteboul and 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 and V. Vianu. Generic computation and its complexity. In *Proceedings of 23rd ACM Symposium on the Theory of Computing*, 1991.
4. F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. A powerful and simple database language. In *Proceedings of International Conference on Very Large Data Bases*, pages 97–105, 1988.
5. R. S. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall International, 1988.
6. V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 9–19. Morgan Kaufmann, August 1991.
7. V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with Sets/Bags/Lists. In *LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming, Madrid, Spain, July 1991*, pages 60–75.

8. V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992*, pages 140–154.
9. L. S. Colby. A recursive algebra for nested relations. *Information Systems*, 15(5):567–582, 1990.
10. L. Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 83:57–69, 1991.
11. P. L. Curien. The $\lambda\rho$-calculus: An abstract framework for environment machines. Technical Report URA 725, Laboratoire d'Informatique, Departement de Mathematiques et d'Informatique, Ecole Normale Superieure, 45 Rue d'Ulm, 75230 Paris Cedex 05, France, 1988.
12. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Combridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1989.
13. M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics*. John Wiley and Sons, Chichester, England, 1990.
14. R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal of Computing*, 15(3):865–886, August 1986.
15. N. Immerman. Length of predicate calculus formulas as a new complexity measure. In *Proceedings of 20th Symposium on Foundations of Computer Science, San Juan, Puerto Rico, October 1979*, pages 337–347.
16. L. Libkin and L. Wong. Some properties of query languages for bags. In *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 97–114.
17. P. Odifreddi. *Classical Recursion Theory*. North Holland, 1989.
18. A. Ohori, P. Buneman, and 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*, pages 46–57, Portland, Oregon, June 1989.
19. J. Paredaens and D. Van Gucht. Converting nested relational algebra expressions into flat algebra expressions. *ACM Transaction on Database Systems*, 17(1):65–93, March 1992.
20. H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.
21. D. Stemple and T. Sheard. A recursive base for database programming. In *LNCS 504: Next Generation Information System Technology*, pages 311–332, Berlin, 1990.
22. D. Suciu and V. Breazu-Tannen. A query language for NC. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 167–178, Minneapolis, Minnesota, May 1994.
23. D. Suciu and J. Paredaens. Any algorithm in the complex object algebra needs exponential space to compute transitive closure. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 201–209, Minneapolis, Minnesota, May 1994.
24. S. J. Thomas and P. C. Fischer. Nested relational structures. In *Advances in Computing Research: The Theory of Databases*, pages 269–307, London, England, 1986. JAI Press.
25. L. Wong. Normal forms and conservative properties for query languages over collection types. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 26–36, Washington, D. C., May 1993.

26. L. Wong. *Querying Nested Collections.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994.

A Proof of Theorem 1

We give the proof in two steps. In the first step, we prove that *sri* is definable in $\mathcal{NRC}(sru)$ augmented with natural numbers and a few operators on natural numbers. In the second step, we show that the extra primitives we have added can be removed. The new primitives that we need are:

$\geq: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$. This is the usual comparison operator on numbers.
$2* : \mathbb{N} \rightarrow \mathbb{N}$. This is the multiply-by-2 function.
$card^s : \{s\} \rightarrow \mathbb{N}$. This function returns the cardinality of the input set.

Now we prove that *sri* can be defined using *sru* without using higher-order functions, when the extra primitives above are available. That is,

Proposition 8. *Let* $sri(i,e) : \{s\} \rightarrow t$ *be well defined. Then it is expressible in* $\mathcal{NRC}(sru, \mathbb{N}, \geq, 2*, card)$.

Proof. We want to define $sri(i,e)(D)$ in terms of *sru* without using higher-order functions. First let $E : \{\{s\}\} \times (\mathbb{N} \times \{\{s\} \times t\})$, $F : \{s\} \rightarrow \{\{s\}\} \times (\mathbb{N} \times \{\{s\} \times t\})$, and $U : (\{\{s\}\} \times (\mathbb{N} \times \{\{s\} \times t\})) \times (\{\{s\}\} \times (\mathbb{N} \times \{\{s\} \times t\})) \rightarrow \{\{s\}\} \times (\mathbb{N} \times \{\{s\} \times t\})$ be defined as follows:

$E \stackrel{\text{def}}{=} (\{\},\ (1, \{(\{\},\ e)\}))$,
$F(X) \stackrel{\text{def}}{=} (\{X\},\ (2, \{(\{\}, e)\}))$, and
$U((A,(m,M)),(B,(n,N))) \stackrel{\text{def}}{=}$ *let* (w,W) *be if* $m \geq n$ *then* (m,M) *else* (n,N) *in if* $card(A \cup B) \geq w$ *then* $(A \cup B,\ (2 * w,\ \{(\{d\} \cup Y,\ i(d,y)) \mid (Y,y) \in W, d \in D\}))$ *else* $(A \cup B, (w, W))$.

Next we define a triple $(A,(m,M))$ to be proper if all three conditions below are satisfied:

1. m is a power of 2. That is, $m = 2^i$.
2. $\lfloor m/2 \rfloor \leq card(A) < m$.
3. $(\{o_1\} \cup \ldots \cup \{o_n\},\ i(o_1, i(o_2, \ldots i(o_n, e) \ldots)) \in M$ iff $o_1 \in D$, ..., $o_n \in D$, and $n = \log(m) - 1$. Note that o_1, ..., o_n are not required to be distinct.

The followings are readily verified:

Claim 1. *Suppose* $(A,(m,M))$ *is proper. Then the cardinality of* A *completely determines* m *and* M.
2. *E is proper.*
3. *$F(X)$ is proper.*
4. *Suppose A and B are proper. Then $U(A,B)$ is proper.*
5. *Suppose A is proper. Then $U(A,E) = U(E,A) = A$.*
6. *Suppose A is proper. Then $U(A,A) = U(A)$.*
7. *Suppose A and B are proper. Then $U(A,B) = U(B,A)$.*
8. *Suppose A, B, and C are proper. Then $U(A,U(B,C)) = U(U(A,B),C)$.*

Proof of claim. We prove only the last item. Let $X \stackrel{\text{def}}{=} (X_1, (X_2, X_3)) \stackrel{\text{def}}{=} U(A, U(B, C))$. Let $Y \stackrel{\text{def}}{=} (Y_1, (Y_2, Y_3)) \stackrel{\text{def}}{=} U(U(A, B), C)$. By the fourth bullet, both X and Y are proper. It is clear that $X_1 = Y_1 = (\pi_1\ A) \cup (\pi_1\ B) \cup (\pi_1\ C)$. Then by the first bullet, $X = Y$ as desired.

Thus $sru(U, F, E)$ is well defined on any subset of $powerset(D)$, where $powerset(D) \stackrel{\text{def}}{=} sru(\lambda X.\{r\ \cup\ s \mid r \in \pi_1\ X,\ s \in \pi_2\ X\},\ \lambda x.\{\{\},\ \{x\}\},\ \{\{\}\})$.

Claim $\{sri(i, e)(D)\} = \{\pi_2 B \mid A \in sru(U, F, E)(powerset(D)),\ B \in \pi_2(\pi_2 A),\ \pi_1 B = D\}$.

Proof of claim. First observe that $sru(U, F, E)(powerset(D)) = (powerset(D), (m, M))$ for some m and M. By being proper, we know that $n \stackrel{\text{def}}{=} card(D) = \log(m) - 1$. Therefore, $(D, i(o_1, i(o_2, \ldots, i(o_n, e) \ldots))) \in M$, where $\{o_1, \ldots, o_n\}$ is the set D. By the idempotent and commutative properties on i, we know that if $(D, x) \in M$, then $x = i(o_1, i(o_2, \ldots, i(o_n, e) \ldots))$. Hence the claim is proved.

However, we still need a function $hd : \{t\} \to t$ such that $hd\{x\} = x$ and $hd(S) = e$ if S is not a singleton set. But this is easy to define: $hd \stackrel{\text{def}}{=} \pi_2(sru(\lambda X.\textit{if } \pi_1(\pi_1\ X) = \textit{false then } \pi_2\ X \textit{ else if } \pi_1(\pi_2\ X) = \textit{false then } \pi_1\ X \textit{ else if } \pi_2(\pi_1\ X) = \pi_2(\pi_2\ X) \textit{ then } \pi_1\ X \textit{ else } (\textit{true}, e),\ \lambda x.(\textit{true}, x),\ (\textit{false},\ e))$. Then $sri(i, e) = hd(\{\pi_2\ B \mid A \in sru(U, F, E)(powerset(D)),\ B \in \pi_3\ A, \pi_1\ B = D\}$. □

The extra primitives above are not critical to the proof of the proposition above. Observe that to compute $sri(i, e)(D)$ using the implementation above, we only need natural numbers 0, ..., $2^{1+card(D)}$. Given the set $D : \{s\}$, the number i can be represented by the set $\mathsf{rep}(i) : \{\{\{\{s\}\}\}\}$ such that $L \in \mathsf{rep}(i)$ iff $L \in powerset(powerset(D))$ and $card(L) \leq i$. For example, $\mathsf{rep}(0)$ is $\{\{\}\}$. Having established our representation of numbers, we demonstrate below the representation of the three new primitives introduced earlier. (We use the shorthand $\mathsf{rep}(\mathbb{N})$ for the type $\{\{\{\{s\}\}\}\}$.)

Proposition 9. *1. There is a function* $\mathsf{rep}(2*) : \mathsf{rep}(\mathbb{N}) \to \mathsf{rep}(\mathbb{N})$ *such that for any* $i \leq 2^{card(D)}$, *it is the case that* $\mathsf{rep}(2*)(\mathsf{rep}(i)) = \mathsf{rep}(2 * i)$.

2. *There is a function* $\mathsf{rep}(\geq) : \mathsf{rep}(\mathbb{N}) \times \mathsf{rep}(\mathbb{N}) \to \mathbb{B}$ *such that for any* $i \leq 2^{1+card(D)}$ *and* $j \leq 2^{1+card(D)}$, *it is the case that* $\mathsf{rep}(\geq)(\mathsf{rep}(i), \mathsf{rep}(j)) = (i \geq j)$.
3. *There is a function* $\mathsf{rep}(card^s \geq) : \{\{\{s\}\}\} \times \mathsf{rep}(\mathbb{N}) \to \mathbb{B}$ *such that for any* $R : \{\{\{s\}\}\} \subseteq powerset(powerset(D))$ *whose cardinality is at most* $2^{card(D)}$ *and for any* $i \leq 2^{1+card(D)}$, *it is the case that* $\mathsf{rep}(card \geq)(R, \mathsf{rep}(i)) = (card(R) \geq i)$.

Proof. Let $X \textit{ or } Y \stackrel{\text{def}}{=} \textit{if } X \textit{ then true else } Y$, $X \textit{ and } Y \stackrel{\text{def}}{=} \textit{if } X \textit{ then } Y \textit{ else false}$, and $X\ isin\ W \stackrel{\text{def}}{=} sru(or,\ \lambda Y.Y = X,\ \textit{false})(W)$. Then

1. $\mathsf{rep}(2*)(W) \stackrel{\text{def}}{=} \{L \mid L \in powerset(powerset(D)),\ L\ isin\ \{l_1 \cup l_2 \mid l_1 \in W,\ l_2 \in W\}\}$.
2. $\mathsf{rep}(\geq)(X,Y) \stackrel{\text{def}}{=} sru(and,\ \lambda y.y\ isin\ X,\ true)(Y)$.
3. $\mathsf{rep}(card \geq)(R,W) \stackrel{\text{def}}{=} R\ isin\ W$. □

Whence,

Theorem 1 *$\mathcal{NRC}(sru, \Sigma) = \mathcal{NRC}(sri, \Sigma)$, for any collection Σ of base types b, primitive functions p, and constants c.*

B Proof of Theorem 2

Theorem 2 *Suppose that every primitive p in the language increases the measure of its input object only by a constant amount; that is, for all $p \in \Sigma$, there is c_p, such that for all complex objects O, $\#(p(O)) \leq c_p + \#(O)$. Then the following hold:*

1. *For every query e in $\mathcal{NRC}(\Sigma)$ or $\mathcal{NRC}(\Sigma, powerset)$, there is a constant c_e such that for every complex object O, we have $\#(e(O)) \leq c_e + \#(O)$.*
2. *For every query e in $\mathcal{NRC}(sru, \Sigma)$, there is are constants c_e and p_e such that for every complex object O, we have $\#(e(O)) \leq c_e \cdot \log^{p_e}(\mathsf{complex}(e, O)) + \#(O)$.*
3. *For every query e in $\mathcal{NRC}(sri, \Sigma)$, there are constants c_e and p_e such that for every complex object O, we have $\#(e(O)) \leq c_e \cdot (\mathsf{complex}(e, O))^{p_e} + \#(O)$.*

Proof sketch. We prove a stronger statement, which we only state for the language $\mathcal{NRC}(sru, \Sigma)$. (It is similar for the other languages.) First extend the definition of the complexity of the evaluation of a query f on a object O, $\mathsf{complex}(f, O)$, to the complexity of the evaluation of a complex object expression e under environment ρ, $\mathsf{complex}(\rho, e)$, and to the complexity of the evaluation of a function expression e applied to a complex object O, under environment ρ, $\mathsf{complex}(\rho, e, O)$. Then the statement we prove is:

1. For any complex object expression e, there are constants c_e and p_e such that for any environment $\rho = \{x_1 := O_1, \ldots, x_n := O_n\}$:

$$\#(O') \leq c_e \cdot \log^{p_e}(\mathsf{complex}(\rho, e)) + \#(\rho)$$

   where $\rho \bullet e \Downarrow O'$. Here $\#(\rho) \stackrel{\text{def}}{=} \#(\{O_1, \ldots, O_n\})$.
2. For any function expression f, there are constants c_f and p_f such that for any environment ρ, and complex object O:

$$\#(O') \leq c_f \cdot \log^{p_f}(\mathsf{complex}(\rho, f, O)) + \max(\#(O), \#(\rho))$$

   where $\rho \bullet f(O) \Downarrow O'$.

In fact, this is the statement of the theorem, in which each expression with free variables is considered as a query having its free variables as inputs. The proof of the statement goes by induction on the structure of e. We illustrate the most difficult case, namely for a function expression f of the form $sru(u,g,e)$. Suppose f is applied to some object $O = \{O_1,\ldots,O_n\}$, namely $\rho \bullet f(O) \Downarrow O'$. Then the last evaluation rule is:

$$\frac{\begin{array}{c}\rho \bullet sru(u,g,e)(\{O_1,\ldots,O_{\lfloor n/2\rfloor}\}) \Downarrow O_{11} \\ \rho \bullet sru(u,g,e)(\{O_{\lfloor n/2\rfloor+1},\ldots,O_n\}) \Downarrow O_{12} \\ \rho \bullet u(O_{11},O_{12}) \Downarrow O\end{array}}{\rho \bullet sru(u,g,e)(\{O_1,\ldots O_n\}) \Downarrow O'} \text{ where } n \geq 2.$$

Going backwards, to the leaves of the evaluation tree, we find:

$$\frac{\begin{array}{c}\rho \bullet sru(u,g,e)(\{O_1,\ldots,O_{\lfloor n/4\rfloor}\}) \Downarrow O_{21} \\ \rho \bullet sru(u,g,e)(\{O_{\lfloor n/4\rfloor+1},\cdots,O_{\lfloor n/2\rfloor}\} \Downarrow O_{22} \\ \rho \bullet u((O_{21},O_{22})) \Downarrow O_{11}\end{array}}{\rho \bullet sru(u,g,e)(\{O_1,\cdots,O_{\lfloor n/2\rfloor}\}) \Downarrow O_{11}}$$

and

$$\frac{\begin{array}{c}\rho \bullet sru(u,g,e)(\{O_{\lfloor n/2\rfloor+1},\cdots,O_{\lfloor 3\cdot n/4\rfloor}\}) \Downarrow O_{23} \\ \rho \bullet sru(u,g,e)(\{O_{\lfloor 3\cdot n/4\rfloor+1},\cdots,O_n\}) \Downarrow O_{24} \\ \rho \bullet u((O_{23},O_{24})) \Downarrow O_{12}\end{array}}{\rho \bullet sru(u,g,e)(\{O_{\lfloor n/2\rfloor+1},\cdots,O_n\}) \Downarrow O_{12}}$$

At each level l, for any intermediate result O_{li}, there are two values at the level $l+1$, say $O_{(l+1)j}$ and $O_{(l+1)(j+1)}$, such that $\rho \bullet u((O_{(l+1)j},O_{(l+1)(j+1)})) \Downarrow O_{li}$. By induction hypothesis, there are some constants p_u and c_u such that:

$$\begin{aligned}\#(O_{li}) &\leq c_u \cdot \log^{p_u}(\mathsf{complex}(\rho,u,(O_{(l+1)j},O_{(l+1)(j+1)}))) \\ &\quad + \max(\#(\rho),\#(O_{(l+1)j}),\#(O_{(l+1)(j+1)})) \\ &\leq c_u \cdot \log^{p_u}(\mathsf{complex}(\rho,f,O)) + \max(\#(\rho),\#(O_{(l+1)j}),\#(O_{(l+1)(j+1)}))\end{aligned}$$

The last inequality is true because the complexity of computing $f(O)$ is at least as large as the complexity of computing $u(O_{(l+1)j},O_{(l+1)(j+1)})$. By induction on the level l, we prove that:

$$\#(O') \leq l \cdot c_u \log^{p_u}(\mathsf{complex}(\rho,f,O)) + \max(\#(\rho),\#(O_{l1}),\#(O_{l2}),\#(O_{l3}),\cdots)$$

Indeed, suppose this holds for some l. Then:

$$\begin{aligned}\#(O') \leq\; & l \cdot c_u \cdot \log^{p_u}(\mathsf{complex}(\rho,f,O)) + \max(\#(\rho), \\ & c_u \cdot \log^{p_u}(\mathsf{complex}(\rho,f,O)) + \max(\#(\rho),\#(O_{(l+1)1}),\#(O_{(l+1)2})), \\ & c_u \cdot \log^{p_u}(\mathsf{complex}(\rho,f,O)) + \max(\#(\rho),\#(O_{(l+1)3}),\#(O_{(l+1)4})),\cdots) \\ \leq\; & l \cdot c_u \cdot \log^{p_u}(\mathsf{complex}(\rho,f,O)) + \\ & c_u \cdot \log^{p_u}(\mathsf{complex}(\rho,f,O)) + \max(\#(\rho),\#(O_{(l+1)1}),\#(O_{(l+1)2}),\cdots)\end{aligned}$$

Now let l be the depth of the balanced tree for computing $\rho \bullet sru(u,g,e)(\{O_1,\ldots,O_n\})$; that is, let $l = \log(n)$. At that level, we apply the rule:

$$\frac{\rho \bullet g(O_k) \Downarrow O_{li}}{\rho \bullet sru(u,g,e)(\{O_k\}) \Downarrow O_{li}} \text{ for some } i$$

Now we apply the induction hypothesis for g, to get:

$$\#(O_{li}) \leq c_g \cdot \log^{p_g}(\mathsf{complex}(\rho, f, O)) + \max(\#(\rho), \#(O_k))$$

Combining the last two inequations with $l = \log(n)$ and noticing that $n \leq \mathsf{complex}(\rho, f, O)$ we get:

$$\begin{aligned}
\#(O') &\leq c_u \cdot \log^{p_u+1}(\mathsf{complex}(\rho, f, O)) + \max(\#(\rho), \#(O_{l1}), \#(O_{l2}), \cdots) \\
&\leq c_u \cdot \log^{p_u+1}(\mathsf{complex}(\rho, f, O)) + \max(\#(\rho), \\
&\qquad c_s \cdot \log^{p_s}(\mathsf{complex}(\rho, f, O)) + \max(\#(\rho), \#(O_1)), \\
&\qquad\quad \vdots, \\
&\qquad c_s \cdot \log^{p_s}(\mathsf{complex}(\rho, f, O)) + \max(\#(\rho), \#(O_n))) \\
&\leq c_u \cdot \log^{p_u+1}(\mathsf{complex}(\rho, f, O)) \\
&\qquad + c_s \cdot \log^{p_s}(\mathsf{complex}(\rho, f, O)) + \max(\#(\rho), \#(O_1), \cdots, \#(O_n)) \\
&= c_u \cdot \log^{p_u+1}(\mathsf{complex}(\rho, f, O)) \\
&\qquad + c_s \cdot \log^{p_s}(\mathsf{complex}(\rho, f, O)) + \max(\#(\rho), \#(\{O_1, \cdots, O_n\})) \\
&\leq c_f \cdot \log^{p_f}(\mathsf{complex}(\rho, f, O)) + \max(\#(\rho), \#(O))
\end{aligned}$$

where $p_f \stackrel{\text{def}}{=} \max(p_u + 1, p_s)$ and $c_f \stackrel{\text{def}}{=} c_u + c_s$. □

C Proof of Corollary 4

Corollary 4 $\mathcal{NRC}^1(sru, \mathbb{N}, 0, succ) < \mathcal{NRC}^1(sri, \mathbb{N}, 0, succ)$.

Proof. The complex objects in this Corollary are considerably simpler than in other parts of this paper. They are now either tuples of atomic objects or are flat relations, as oppose to nested relations in the general case. We use A and B to denote these simple types.

We define a notion of limiting size $\mathsf{limsize}(A, n)$ of type A, where $ht(A) \leq 1$, with respect to n as follows:

$\mathsf{limsize}(\mathbb{N}, n) = 2 + n$;
$\mathsf{limsize}(unit, n) = 2$;
$\mathsf{limsize}(A \times B, n) = \mathsf{limsize}(A, n) \cdot \mathsf{limsize}(B, n)$; and
$\mathsf{limsize}(\{A\}, n) = \mathsf{limsize}(A, n)$. Note that $ht(A) = 0$ in this case.

And we define the measure of an object of type $\mathbb{N}$ to be $\#(succ^n(0)) \stackrel{\text{def}}{=} n$. Then the corollary follows from the claim below.

Claim *1.* $\mathsf{limsize}(A, n) \leq n^p$, *where* $2 \leq n$ *and* p *is a constant depending only on* A.
2. Suppose $O : A$ *is a complex object such that* $\#(O) \leq n$, *where* $2 \leq n$. *Then* $\mathsf{size}(O) \leq \mathsf{limsize}(A, n)$.

3. *There is a function* $f : \{\mathbb{N}\} \to \mathbb{N}$ *in* $\mathcal{NRC}^1(sri, \mathbb{N}, 0, succ)$ *such that* $\#(f(O)) \geq 2^{card(O)}$.
4. *There is no function* $f : \{\mathbb{N}\} \to \mathbb{N}$ *in* $\mathcal{NRC}^1(sru, \mathbb{N}, 0, succ)$ *such that* $\#(f(O)) \geq 2^{card(O)}$.

Proof of claim. We prove only the last item, as the other three are easy. Let G be any function of type $\{\mathbb{N}\} \to \mathbb{N}$ defined in $\mathcal{NRC}^1(sru, \mathbb{N}, 0, succ)$. Let A be a type in the type derivation of G so that $\mathsf{limsize}(A, n)$ is maximum for all $n \geq 2$ (this is possible since the definition of $\mathsf{limsize}$ is monotone in its second argument). Let $N = \#(O) \geq 2$. Let M be the maximum measure over all intermediate data in all executions of $G(O)$. Let $O' : A'$ be the intermediate data whose size is greatest in all executions of $G(O)$. Then the corollary follows immediately form the following claim.

$$\begin{aligned}
\#(G(O)) &\leq a \cdot \log^b(\mathsf{size}(O')) + N && \text{Theorem 2} \\
&\leq a \cdot \log^b(\mathsf{limsize}(A', M)) + N && \text{Claim 2} \\
&\leq a \cdot \log^b(\mathsf{limsize}(A, M)) + N && \text{Choice of } A \\
&\leq a \cdot \log^b(M^c) + N && \text{Claim 1}
\end{aligned}$$

Clearly,

$$\begin{aligned}
M &\leq a \cdot \log^b(M^c) + N \\
&= d \cdot \log^b(M) + N \quad \text{With } d = a \cdot c^b
\end{aligned}$$

But $M = N + D$ for some D,

$$N + D \leq d \cdot \log^b(N + D) + N$$

Hence

$$D \leq d \cdot \log^b(N + D)$$

So

$$2^{\sqrt[b]{D/d}} - D \leq N$$

It is clear that D cannot be exponential with respect to N. Since $\#(G(O)) \leq M$, as M was chosen to be the maximum measure over all intermediate data, we conclude that $\#(G(O))$ cannot be exponential with respect to N. This proves the claim and the corollary. □

D Proof of Proposition 5

Assume that Σ is a signature containing some constants $c : b$ and first-order primitives $p : s \to t$ and nothing else. Each $p : s \to t$ must be internal in the sense of Hull [14]. That is, we require that $restrict^t_{b_0}(p(o)) = p(o)$ for all finite $b_0 \subseteq b$ and $o = restrict^s_{b_0}(o)$.

Proposition 5 *Let $e : S$ be an expression of $\mathcal{HNRC}(sri, \Sigma)$ with $x_1 : S_1$, ..., $x_n : S_n$ as free variables. Let b_0 be a finite subset of b that includes all constants $c : b$ occurring in e. Then there is an expression $rel(e) : rel(S)$ of $\mathcal{NRC}(sri, \Sigma)$ with $\bar{x_1} : rel(S_1)$, ..., $\bar{x_n} : rel(S_n)$ as free variables such that for all $O_1 : S_1$, ..., $O_n : S_n$, where $restrict^{S_i}_{b_0}(O_i) : rel(S_i)$ is in $restrict_{b_0}(S_i)$ for each i, then $restrict^{S}_{b_0}(e[O_1/x_1, ..., O_n/x_n])$ is defined and equals $rel(e)[restrict^{S_1}_{b_0}(O_1)/\bar{x_1}$, ..., $restrict^{S_n}_{b_0}(O_n)/\bar{x_n}]$.*

Proof. It suffices to defined $rel(e)$ by induction on the structure of e as follows.

$rel(true) \stackrel{\text{def}}{=} true$
$rel(false) \stackrel{\text{def}}{=} false$
$rel(if\ e_1\ then\ e_2\ else\ e_3) \stackrel{\text{def}}{=} if\ rel(e_1)\ then\ rel(e_2)\ else\ rel(e_3)\ rel() \stackrel{\text{def}}{=} ()$
$rel(c) \stackrel{\text{def}}{=} c$
$rel(\pi_i\ e) \stackrel{\text{def}}{=} \pi_i(rel(e))$
$rel(e_1, e_2) \stackrel{\text{def}}{=} (rel(e_1), rel(e_2))$
$rel(e_1 = e_2) \stackrel{\text{def}}{=} rel(e_1) = rel(e_2)$
$rel(x) \stackrel{\text{def}}{=} \bar{x}$
$rel(\{\}) \stackrel{\text{def}}{=} \{\}$
$rel(\{e\}) \stackrel{\text{def}}{=} \{rel(e)\}$
$rel(e_1\ \cup\ e_2) \stackrel{\text{def}}{=} rel(e_1) \cup rel(e_2)$
$rel(p : s \to t) \stackrel{\text{def}}{=} \{(x, y) \mid (x, y) \in restrict_{b_0}(s \times t),\ y = p(x)\}$
$rel(\lambda x.e : S \to T) \stackrel{\text{def}}{=} \{(\bar{x}, rel(e)) \mid \bar{x} \in restrict_{b_0}(S)\}$.
$rel(e_1\ e_2) \stackrel{\text{def}}{=} app(rel(e_1), rel(e_2))$, where $app(F, E) \stackrel{\text{def}}{=} hd(\bigcup\{if\ E = x\ then\ \{y\}\ else\ \{\} \mid (x, y) \in F\})$
$rel(sri(i, e) : \{s\} \to t) \stackrel{\text{def}}{=} SRI(rel(i), rel(e))$, where $SRI(I, E) \stackrel{\text{def}}{=} \{(X, sri(\lambda x.app(I, x),\ E)(X)) \mid X \in restrict_{b_0}(\{s\})\}$.

□

This article was processed using the LaTeX macro package with LLNCS style