Comprehending nulls

Comprehending nulls

James Cheney
University of Edinburgh
jcheney@inf.ed.ac.uk

Wilmer Ricciotti
University of Edinburgh
research@wilmer-ricciotti.net

ABSTRACT

The Nested Relational Calculus (NRC) has been an influential high-level query language, providing power and flexibility while still allowing translation to standard SQL queries. It has also been used as a basis for language-integrated query in programming languages such as F#, Scala, and Links. However, SQL's treatment of incomplete information, using nulls and three-valued logic, is not compatible with 'standard' NRC based on two-valued logic. Nulls are widely used in practice for incomplete data, but the question of how to accommodate SQL-style nulls and incomplete information in NRC, or integrate such queries into a typed programming language, appears not to have been studied thoroughly. In this paper we consider two approaches: an *explicit* approach in which option types are used to represent (possibly) nullable primitive types, and an *implicit* approach in which types are treated as possibly-null by default. We give translations relating the implicit and explicit approaches, discuss handling nulls in language integration, and sketch extensions of normalization and conservativity results.

1 INTRODUCTION

The Nested Relational Calculus (NRC) [2] is a high-level query language providing operations for collections (sets, bags, lists, etc.), especially *comprehensions*. In contrast to standard query languages such as SQL, NRC queries can be freely composed and can construct values with nesting of record and collection types, making it natural to use for database programming and query integration in high-level functional languages [3, 8, 15, 20]. Despite this added flexibility, NRC queries are no more expressive than flat relational queries when transforming flat inputs to flat outputs [22]. This property, called *conservativity*, is the basis for rewriting algorithms that map NRC queries over flat data to SQL queries.

From the early years of the development of the relational data model and associated query languages, the importance of supporting incomplete information has been clearly recognized. Codd [6] made an early proposal allowing field values to be "null", or absent/missing, extending primitive operations on these values to propagate nulls, and extending predicates to have three-valued semantics with a third truth value, "unknown". Despite criticism [10], this approach is standard and widely used in SQL, although these features are also easily misunderstood and result in counterintuitive behavior that can lead to subtle bugs [11]. Nevertheless, almost all real databases and applications involve nulls, so it is important for language-integrated query mechanisms to support them.

Most presentations of NRC and related languages eschew nulls: base types include integers, booleans, strings, etc. as understood in most typed programming languages, in which there is no special null value indicating an absent piece of data. This makes NRC a good fit for integrating database queries into an ambient typed language, but a poor fit for interfacing with actual incomplete data. Moreover, while SQL's approach to nulls is imperfect, a language-integrated query system should still be able deal with them.

In this short paper, we investigate the design issues that arise when we add null values to NRC, highlight technical issues whose solutions are straightforward or already known, and outline open questions. In particular we consider the following issues:

(1) Should nulls be treated implicitly (like in SQL) or explicitly (like option values in functional languages)?
(2) Should nulls be available at any type, or just at base types?
(3) Do classical results needed for translating NRC queries to SQL continue to hold in the presence of nulls?

Design considerations. Our goal is to reconcile the implicit treatment of nulls in a typical database query language (e.g. SQL) with a typed, functional host language that lacks nulls. We first give a toy example and discuss how it is handled currently in three settings: Links [8], Scala's Quill library [15], and in LINQ in F# [3, 20].

Suppose we have a table containing diseases, each with identifier (integer), name (string), and type (integer). The identifier and name are required (i.e. non-nullable) but the type is optional and nullable (some new diseases might not yet have a known type). To produce a web page showing information related to diseases with a given name, we would execute a query such as

```
SELECT * FROM diseases WHERE name = 'covid-19'
```

In Links, until recently, attempting to execute queries that attempted to read NULLs from the type field would simply fail, because the NULL value was not expected by the code that processes query results. Currently, Links allows to set a single global default value to use in place of NULL for integer fields.

In F#, in contrast, nullable fields in database tables or query results are given a different type: `Nullable<T>`. A value of type `T` can be implicitly coerced to `Nullable<T>`, and this type also includes a null value. Whether a `Nullable<T>` is null or not can be tested by checking the Boolean field `HasValue`, and if present the value can be extracted from the `Value` field. Requesting the value of a null yields an exception. Primitive operators such as addition and equality (+, =) are lifted to nullable versions (?+, ?=) that propagate nulls like SQL does: if any input is null then the result is null.

In Quill, nullable fields are given option types, and Scala overloading and convenient operations on option types can be used to make it easier to write queries involving such optional data.

Obviously, the Links solution is little better than a hack: if we wanted to deal with nulls of other base types, we would have to provide a default value, and it isn't clear that using a single global default in place of null values of each type is sensible. On the other hand, the F# and Quill approaches appear to work reasonably well in practice, but rely on implicit coercions and exceptions, and still require programmers to be conscious of which fields are nullable.

If we look beyond the simple scenario above in which we are just retrieving data (possibly including NULLs) from the database,

the situation becomes a bit more complicated. In SQL, as mentioned above, most primitive operations are defined so that the result is null if any input is null; some operations such as logical connectives and null tests depart from this pattern. Null boolean values, also called unknowns, provide a third truth value, resulting in behavior that can be counterintuitive. Moreover, it is not clear that query rewriting laws that are valid in standard two-valued logic still hold, calling into question whether the rewriting strategy used in Links to normalize and generate SQL from NRC queries is still viable. We should also note that neither F#'s handling of nulls via nullable types nor Quill's treatment using option types is supported by any formal analysis like that for basic language-integrated query [3], so it is unclear what formal guarantees these approaches have.

A final consideration, which is not strictly necessary to deal with the problem of incomplete data in SQL, but seems natural to consider in a nested relational setting, is whether null values ought to be considered only for base types (integers, strings etc.) or for the composite NRC types including records and collection types. The latter approach seems more uniform and more in the spirit of NRC, but leads immediately to the question whether allowing nulls at composite types increases expressiveness, or whether the classical results on conservativity still hold.

Summing up, we would like to reconcile database queries involving nulls with typed host languages so that:

(1) Null values are available at all types and query results including nulls can be translated to host language values.
(2) Query expressions can be written as in SQL: e.g. primitive operations apply uniformly to nullable and nonnullable fields
(3) Query expressions admit normalization rules similar to those for plain NRC, enabling translation to SQL.

Moreover, we would like to accomplish these goals in a way that makes programming as easy as possible in common cases, and that avoids reliance on advanced programming language features as much as possible. We note again that none of the approaches we are aware of in Links, F# or Quill satisfy all three criteria.

2 BACKGROUND

We will employ the following syntax for NRC:

$$\begin{array}{llll} \textbf{Types} & \sigma, \tau & ::= & b \mid \langle \overrightarrow{\ell : \sigma} \rangle \mid \{\sigma\} \\ \textbf{Terms} & M, N & ::= & x \mid c \mid f(\overrightarrow{M}) \mid \langle \overrightarrow{\ell = M} \rangle \mid M.\ell \\ & & \mid & \emptyset \mid \{M\} \mid M \cup N \mid \bigcup\{M | x \leftarrow N\} \\ & & \mid & \mathtt{empty}(M) \mid \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 \end{array}$$

The base types b include integers, strings, booleans, floating-point numbers, dates, etc. Constants c and primitive operations f operate on base types, and include Boolean constants and logical connectives $\mathtt{true}, \mathtt{false}, \wedge, \vee, \neg$. Record types are written $\langle \overrightarrow{\ell : \sigma} \rangle$ with records constructed as $\langle \overrightarrow{\ell = M} \rangle$ and field projection written $M.\ell$. We consider a single set collection type written $\{\sigma\}$. The expressions involving collections include the empty collection $\emptyset$, singleton $\{M\}$, union $M \cup N$, and comprehension $\bigcup\{M | x \leftarrow N\}$ where M is evaluated repeatedly with x bound to elements of N and the resulting collections are unioned. Finally, the conditional $\mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2$ has the standard behavior.

We write $M\ \mathtt{where}\ N$ to abbreviate $\mathtt{if}\ N\ \mathtt{then}\ M\ \mathtt{else}\ \emptyset$, i.e. return M if N holds, otherwise $\emptyset$. A general comprehension (where M may have any type) $\{M | x_1 \leftarrow N_1, \ldots x_k \leftarrow N_k\ \mathtt{where}\ P\}$, is syntactic sugar for $\bigcup\{\cdots\bigcup\{\{M\}\ \mathtt{where}\ P | x_k \leftarrow N_k\}\cdots | x_1 \leftarrow N_1\}$. Such comprehensions correspond to conjunctive SQL queries.

The (largely standard) type system and common rewriting rules for evaluating and translating queries in this variant of NRC are included in the appendix.

3 EXPLICIT NULLS

We extend the core NRC with explicit nulls, calling this calculus $\mathrm{NRC}_{\mathrm{opt}}$, as follows.

$$\begin{array}{llll} \textbf{Types} & \sigma, \tau & ::= & \cdots \mid \tau? \\ \textbf{Terms} & M, N & ::= & \cdots \mid \mathtt{none} \mid \mathtt{some}(M) \\ & & \mid & \mathtt{case}\ M\ \mathtt{of}\ (\mathtt{none} \Rightarrow N_1 \mid \mathtt{some}(x) \Rightarrow N_2) \end{array}$$

We introduce a new type $\tau?$ (pronounced "τ option") whose values are $\mathtt{none}$ and $\mathtt{some}(V)$ where V is of type τ. The elimination form for $\tau?$ is the case construct $\mathtt{case}\ M\ \mathtt{of}\ (\mathtt{none} \Rightarrow N_1 \mid \mathtt{some}(x) \Rightarrow N_2)$ which inspects M, and returns N_1 if M is $\mathtt{none}$ and $N_2[V/x]$ if M is $\mathtt{some}(V)$. Intuitively, optional values correspond to nullable values in SQL. Thus, given a table with some nullable fields, these fields can be represented using option types, whereas non-nullable fields are represented using an ordinary type.

The semantics of option types and expressions is standard:

$$\begin{array}{rcl} \mathtt{case}\ \mathtt{none}\ \mathtt{of}\ (\mathtt{none} \Rightarrow N_1 \mid \mathtt{some}(x) \Rightarrow N_2) & \rightsquigarrow & N_1 \\ \mathtt{case}\ \mathtt{some}(M)\ \mathtt{of}\ (\mathtt{none} \Rightarrow N_1 \mid \mathtt{some}(x) \Rightarrow N_2) & \rightsquigarrow & N_2[M/x] \end{array}$$

Thus, $\mathrm{NRC}_{\mathrm{opt}}$ essentially models the Quill approach, but the advanced features of Scala that make it more palatable are absent.

4 IMPLICIT NULLS

The explicit calculus $\mathrm{NRC}_{\mathrm{opt}}$ provides a correct, and implementable, strategy for handling incomplete information: we simply map nullable types in database tables to option types, and require the query to perform any case analysis. However, making nulls explicit using option types is not cost-free: in the unfortunately all-too-common case where the database schema does not specify fields as nonnull (even if they are in practice never null), the programmer is forced to program defensively by handling both the $\mathtt{none}$ and $\mathtt{some}()$ cases for each field used by the query. This is especially painful when performing primitive operations on multiple nullable values: for example to simulate SQL's behavior when adding two integers that might be null, we need to perform case analysis on the first one, then a sub-case analysis on the second one.

In this section we consider an alternative approach, $\mathrm{NRC}_{\mathrm{null}}$, in which all base types are treated as including an extra value $\mathtt{null}$. The semantics of primitive operations is augmented to handle null value inputs; in most cases, if any input value is null then the result is null. The exceptions are the logical connectives, which are instead equipped with three-valued semantics (e.g. $\mathtt{false} \wedge \mathtt{null} = \mathtt{false}$), and operations such as $\mathtt{isNull}(M)$ that inspect a possibly-null primitive value and test whether it is null.

The syntax of NRC_{null} is NRC extended with a null constant and with a nullness test, as follows. We assume the presence of primitive operations including at least the logical connectives $\wedge, \vee, \neg$.

$$\textbf{Terms} \quad M, N \quad ::= \quad \cdots \mid \texttt{null} \mid \texttt{isNull}(M)$$

We do not allow nulls at record or collection types. For collection types in particular, the expected behavior of nulls is unclear. The semantics of logical connectives is three-valued, as in SQL. The semantics of other primitive operations is strict: if any argument is null then the result is too, otherwise the primitive operation is performed on the non-null inputs. Finally, if the Boolean in a `where` statement is null, then the statement evaluates to an empty collection (similarly to `false` and contrary to `true`). This behavior can be specified by adding the following rewriting rules to the standard NRC ones:

$$\begin{array}{llll} \texttt{isNull(null)} & \rightsquigarrow & \texttt{true} & \texttt{isNull}(c) \rightsquigarrow \texttt{false} \\ f(\ldots\texttt{null}\ldots) & \rightsquigarrow & \texttt{null} & M\ \texttt{where null} \rightsquigarrow \emptyset \end{array}$$

5 TRANSLATIONS

The implicit and explicit approaches have complementary advantages. NRC_{opt} is essentially a special case of the nested relational calculus with binary sum types. However, if many fields are nullable, writing queries in NRC_{opt} is excruciating. On the other hand, NRC_{null} seems easier to relate to plain SQL queries, and writing queries that operate over possibly-null values is more straightforward (albeit with the same pitfalls as SQL), but normalization results for NRC with implicit nulls do not follow immediately from prior work. We consider translations in each direction.

From NRC_{opt} *to* NRC_{null}. The main issue arising in this translation is the fact that option types can be nested inside other type constructors, including options: for example $(\textbf{int}? \times \textbf{bool})?$ represents an optional pair the first element of which is also optional. To deal with this generality, we translate options and cases as follows:

$$\begin{array}{rcl} [\![\tau?]\!] & = & \langle \mathsf{isnull} : \textbf{bool}, \mathsf{val} : [\![\tau]\!] \rangle \\ [\![\texttt{none}]\!] & = & \langle \mathsf{isnull} = \texttt{true}, \mathsf{val} = d_{[\![\tau]\!]} \rangle \\ [\![\texttt{some}(M)]\!] & = & \langle \mathsf{isnull} = \texttt{false}, \mathsf{val} = [\![M]\!] \rangle \\ \left[\!\!\left[\begin{array}{l} \texttt{case } M \\ \texttt{ of (none} \Rightarrow N_1 \\ \texttt{ | some}(x) \Rightarrow N_2) \end{array} \right]\!\!\right] & = & \begin{array}{l} \texttt{if } M.\mathsf{isnull} \\ \texttt{ then } [\![N_1]\!] \\ \texttt{ else } [\![N_2]\!][[\![M]\!].\mathsf{val}/x] \end{array} \end{array}$$

where d_τ is a default value of type τ. Note that nulls, $\texttt{isNull}(-)$ and other null-sensitive primitive operations are not needed to handle options, assuming that there are constants of each base type in NRC_{opt}: this translation actually maps NRC_{opt} to plain NRC.

From NRC_{null} *to* NRC_{opt}. Types are translated as follows:

$$[\![b]\!] = b? \qquad [\![\langle \overrightarrow{\ell : \tau} \rangle]\!] = \langle \overrightarrow{\ell : [\![\tau]\!]} \rangle \qquad [\![\{\tau\}]\!] = \{[\![\tau]\!]\}$$

The most interesting cases of the term translation are:

$$\begin{array}{rcl} [\![c]\!] & = & \texttt{some}(c) \\ [\![f(M_1, \ldots, M_n)]\!] & = & f^*([\![M_1]\!], \ldots, [\![M_n]\!]) \\ [\![\texttt{null}]\!] & = & \texttt{none} \\ [\![\texttt{isNull}(M)]\!] & = & [\![M]\!] = \texttt{none} \\ [\![\texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2]\!] & = & \texttt{if isTrue}([\![M]\!]) \texttt{ then } [\![N_1]\!] \texttt{ else } [\![N_2]\!] \end{array}$$

Here f^* is the primitive operation f lifted to apply to options, i.e. $f^*(\texttt{some}(v_1), \ldots, \texttt{some}(v_n)) = \texttt{some}(f(v_1, \ldots, v_n))$ and otherwise $f^*(\ldots \texttt{none} \ldots) = \texttt{none}$. These operations are definable in NRC_{opt}, as are the other null-sensitive operations such as equality and logical connectives. Conditionals must be translated so that the then-branch is executed only if the test is true, and the else-branch if the test is false or null. To ensure this we use the auxiliary operation $\texttt{isTrue}(x) = \texttt{case } x \texttt{ of } (\texttt{none} \Rightarrow \texttt{false} \mid \texttt{some}(y) \Rightarrow y)$.

6 HANDLING NULLS IN QUERY RESULTS

The translations above establish that NRC_{opt} and NRC_{null} are equally expressive (and equally expressive to NRC provided all base types have default values). In principle one could allow programmers to write queries in NRC_{null}, generate and evaluate the corresponding SQL queries, and translate the results at the end to host language values involving options. How can we make it easy to work with these results in a host language where field types do not have nulls?

Nullable type tracking. This idea is a slightly strengthened form of F#'s approach. The type system could be extended to track nullability information in queries, and using this information try to minimize the amount of optional tagging that must be added. In particular, this approach could cope with the overloaded behavior of primitive operations on nulls, by giving them types that indicate that the result may be null only if one of the inputs may be null; if all inputs are nonnull then so is the result. This approach could be encoded using a sufficiently rich type system, e.g. dependent types or Haskell's type families. However, if schemas lack accurate information about nullability, any benefits may be limited.

Null handlers. This idea is loosely inspired by the common language feature of exception handling, and by Quill's pragmatic approach to dealing with optional values inside queries. Given a query returning flat records in NRC_{null}, we could consider a small domain-specific language of *null handlers* that specify how to map the result to an NRC_{opt} value. A null handler is a record of instructions defining what to do with each possibly-null field:

(1) optional: return an option value
(2) required: skip this record if this field is null
(3) default v: return default value v if null

Syntactic sugar for declaring multiple fields optional or required may also be useful. Of course, it is possible to provide any other desired behavior by returning all nullable results as optional values. If nulls are tracked by the type system, then fields that are certainly nonnull do not need to be mentioned.

For example, the disease table query from Section 1 could have (among others) two handlers:

$$\langle \mathsf{id} : \mathsf{required}, \mathsf{name} : \mathsf{required}, \mathsf{type} : \mathsf{default}\ -1 \rangle$$
$$\langle \mathsf{id} : \mathsf{required}, \mathsf{name} : \mathsf{required}, \mathsf{type} : \mathsf{required} \rangle$$

The first one will use -1, an invalid type value, if a type field is null, while the second will skip any records that contain null type fields. Nulls in the id and name fields could also lead to records being dropped, but should not occur according to the schema. These handlers can be desugared to case analyses using `isNull()` (on the database side) or case (in the host language). By desugaring

to database-side case analyses, the handling can be performed in the database, possibly saving effort.

7 RELATED AND FUTURE WORK

Though nulls and incomplete information have been studied extensively for traditional query languages over flat data (see Libkin [13] for a recent overview), these features appear to have attracted limited interest in the setting of nested relational calculus or complex object query languages. The only work in this direction we know of is from the early years of 'non-first-normal-form' databases [12, 19]. Roth et al. [19] studied nested relations with several variants of nulls, including no-information, does-not-exist, and unknown, while Levene and Loizou [12] considered only a single 'no-information' null, however neither of these approaches corresponds exactly to the treatment of nulls in SQL, as formalized recently by Guagliardo and Libkin [11].

Sum types (of which $\tau?$ is a special case) were studied in an NRC setting by Wong [22]. Wong showed normalization and conservative extension properties hold in the presence of sums and later Giorgidze et al. [9] showed that nonrecursive algebraic data types (i.e. n-ary labeled sums) can be implemented in NRC by mapping such datatypes to nested collections. However, for the purposes of normalizing queries and generating SQL, the latter approach has the disadvantage that query results would use nested collections to represent options, requiring a further flattening or shredding step possibly resulting in executing several SQL queries [4, 21], which is not needed in our translation. General sum types can also be simulated using options, e.g. by representing $\tau + \sigma$ as $\langle L : \tau?, R : \sigma? \rangle$. Implementing sum types using nulls is possible future work.

In this paper we have focused on nulls in a conventional NRC with a single collection type, e.g. homogeneous sets or multisets. In SQL, which contains operators with both set and multiset semantics, as well as grouping and aggregation, nulls interact with several other features, such as multiset difference and aggregation, often in counterintuitive ways [1, 11]. Our focus has been on semantics of NRC queries in the presence of nulls. We conjecture that normalization and conservativity results hold for $\mathrm{NRC}_{\mathrm{null}}$ and $\mathrm{NRC}_{\mathrm{opt}}$ facilitating their translation to flat SQL queries. We are also interested in generalizing our treatment of nulls to queries over heterogeneous (set/bag) collections [16], higher-order functions [7, 17], grouping and aggregation [14], and to shredding queries that produce nested results into multiple SQL queries [4, 18] and in extending $\mathrm{NRC}_{\mathrm{null}}$ to allow nulls at record and collection types. Such extensions seem possible but not necessarily straightforward. For example, should a union of a null collection with another be null, or should the result retain partial knowledge about the known elements?

8 CONCLUSIONS

Incomplete information is needed in most real database situations. While incomplete information has been studied extensively both in theory (e.g. certain answer semantics [13]) and practice (e.g SQL's pragmatic, but complex treatment of nulls and three-valued logic [11]), almost all such work has focused on conventional, flat relational data and queries, not nested relations. This gap in the literature is particularly noticeable where clean query languages such as NRC are used to embed SQL queries safely into an ambient typed programming language, as in Links, F#, or Quill. In this short paper, we have outlined the main issues and design considerations we think are important for a satisfactory solution to this problem. We have also outlined some initial technical steps towards a solution.

ACKNOWLEDGMENTS

This work was supported by ERC Consolidator Grant Skye (grant number ERC 682315), and by an ISCF Metrology Fellowship grant provided by the UK government's Department for Business, Energy and Industrial Strategy (BEIS).

REFERENCES

[1] V. Benzaken and E. Contejean. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *CPP*, 2019.
[2] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1), 1995.
[3] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, 2013.
[4] J. Cheney, S. Lindley, and P. Wadler. Query shredding: efficient relational evaluation of queries over nested multisets. In *SIGMOD*. ACM, 2014.
[5] J. Cheney and W. Ricciotti. Comprehending nulls (extended version). Technical report, arXiv, 2021.
[6] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, 1979.
[7] E. Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL*, 2009.
[8] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: web programming without tiers. In *FMCO*, 2007.
[9] G. Giorgidze, T. Grust, A. Ulrich, and J. Weijers. Algebraic data types for language-integrated queries. In *DDFP*, pages 5–10, 2013.
[10] J. Grant. Null values in SQL. *SIGMOD Rec.*, 37(3):23–25, Sept. 2008.
[11] P. Guagliardo and L. Libkin. A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 2017.
[12] M. Levene and G. Loizou. Semantics for null extended nested relations. *ACM Trans. Database Syst.*, 18(3):414–459, 1993.
[13] L. Libkin. Incomplete data: what went wrong, and how to fix it. In *PODS*, pages 1–13, 2014.
[14] R. Okura and Y. Kameyama. Language-integrated query with nested data structures and grouping. In *FLOPS*, pages 139–158, 2020.
[15] Quill: Compile-time language integrated queries for Scala. Open source project. https://github.com/getquill/quill.
[16] W. Ricciotti and J. Cheney. Mixing set and bag semantics. In *DBPL*, pages 70–73, 2019.
[17] W. Ricciotti and J. Cheney. Strongly normalizing higher-order relational queries. In *FSCD*, pages 28:1–28:22, 2020.
[18] W. Ricciotti and J. Cheney. Query lifting - language-integrated query for heterogeneous nested collections. In *ESOP*, pages 579–606, 2021.
[19] M. A. Roth, H. F. Korth, and A. Silberschatz. Null values in nested relational databases. *Acta Informatica*, 26(7):615–642, 1989.
[20] D. Syme. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *ML Workshop*, 2006.
[21] A. Ulrich. *Query Flattening and the Nested Data Parallelism Paradigm.* PhD thesis, University of Tübingen, Germany, 2019.
[22] L. Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3), 1996.

A TYPING RULES

A.1 Rules for NRC

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Sigma(c) = b}{\Gamma \vdash c : b}$$

$$\frac{\Sigma(f) = \overrightarrow{b_n} \Rightarrow b' \qquad (\Gamma \vdash M_i : b_i)_{i=1,\ldots,n}}{\Gamma \vdash f(\overrightarrow{M_n}) : b'}$$

$$\frac{(\Gamma \vdash M_i : \tau_i)_{i=1,\ldots,n}}{\Gamma \vdash \langle \overrightarrow{\ell_n = M_n} \rangle : \langle \overrightarrow{\ell_n : \tau_n} \rangle} \qquad \frac{\Gamma \vdash M : \langle \overrightarrow{\ell_n : \tau_n} \rangle \qquad i \in \{1, \ldots, n\}}{\Gamma \vdash M.\ell_i : \tau_i}$$

$$\frac{}{\Gamma \vdash \emptyset : \{\tau\}} \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \{M\} : \{\tau\}} \qquad \frac{\Gamma \vdash M : \{\tau\} \qquad \Gamma \vdash N : \{\tau\}}{\Gamma \vdash M \cup N : \{\tau\}}$$

$$\frac{\Gamma, x : \sigma \vdash M : \{\tau\} \qquad \Gamma \vdash N : \{\sigma\}}{\Gamma \vdash \bigcup\{M | x \leftarrow N\} : \{\tau\}} \qquad \frac{\Gamma \vdash M : \{\tau\}}{\Gamma \vdash \texttt{empty}(M) : \textbf{bool}}$$

$$\frac{\Gamma \vdash M : \textbf{bool} \qquad \Gamma \vdash N_1 : \tau \qquad \Gamma \vdash N_2 : \tau}{\Gamma \vdash \texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 : \tau}$$

A.2 Additional rules for $\mathrm{NRC}_{\mathrm{opt}}$

For $\mathrm{NRC}_{\mathrm{opt}}$ the following typing rules are added to those of NRC:

$$\frac{}{\Gamma \vdash \texttt{none} : \tau?} \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \texttt{some}(M) : \tau?}$$

$$\frac{\Gamma \vdash M : \tau? \qquad \Gamma \vdash N_1 : \sigma \qquad \Gamma, x : \tau \vdash N_2 : \sigma}{\Gamma \vdash \texttt{case } M \texttt{ of } (\texttt{none} \Rightarrow N_1 \mid \texttt{some}(x) \Rightarrow N_2) : \sigma}$$

A.3 Additional rules for $\mathrm{NRC}_{\mathrm{null}}$

For $\mathrm{NRC}_{\mathrm{null}}$ the following typing rules are added to those of NRC:

$$\frac{}{\Gamma \vdash \texttt{null} : b} \qquad \frac{\Gamma \vdash M : b}{\Gamma \vdash \texttt{isNull}(M) : \textbf{bool}}$$

B REWRITE RULES

B.1 Common rules

$$\langle \ldots, \ell = M, \ldots \rangle.\ell \rightsquigarrow M \qquad f(\overrightarrow{V}) \rightsquigarrow [\![f]\!](\overrightarrow{V})$$

$$\bigcup\{\emptyset | x \leftarrow M\} \rightsquigarrow \emptyset \qquad \bigcup\{M | x \leftarrow \emptyset\} \rightsquigarrow \emptyset$$

$$\bigcup\{M | x \leftarrow \{N\}\} \rightsquigarrow M[N/x]$$

$$\bigcup\{M \cup N | x \leftarrow R\} \rightsquigarrow \bigcup\{M | x \leftarrow R\} \cup \bigcup\{N | x \leftarrow R\}$$

$$\bigcup\{M | x \leftarrow N \cup R\} \rightsquigarrow \bigcup\{M | x \leftarrow N\} \cup \bigcup\{M | x \leftarrow R\}$$

$$\bigcup\{M | y \leftarrow \bigcup\{R | x \leftarrow N\}\} \rightsquigarrow \bigcup\{\bigcup\{M | y \leftarrow R\} | x \leftarrow N\} \quad (\text{if } x \notin \mathrm{FV}(M))$$

$$\bigcup\{M | x \leftarrow N \texttt{ where } L\} \rightsquigarrow \bigcup\{M | x \leftarrow N\} \texttt{ where } L$$

$$M \texttt{ where true} \rightsquigarrow M \qquad M \texttt{ where false} \rightsquigarrow \emptyset$$

$$\emptyset \texttt{ where } L \rightsquigarrow \emptyset$$

$$(M \cup N) \texttt{ where } L \rightsquigarrow (M \texttt{ where } L) \cup (N \texttt{ where } L)$$

$$\bigcup\{M | x \leftarrow N\} \texttt{ where } L \rightsquigarrow \bigcup\{M \texttt{ where } L | x \leftarrow N\} \quad (\text{if } x \notin \mathrm{FV}(L))$$

$$(M \texttt{ where } L_1) \texttt{ where } L_2 \rightsquigarrow M \texttt{ where } (L_1 \wedge L_2)$$

$$\texttt{empty } M \rightsquigarrow \texttt{empty } (\bigcup\{\langle\rangle | x \leftarrow M\}) \quad (\text{if } M \text{ is not relation-typed})$$

$$\texttt{if } L \texttt{ then } M \texttt{ else } N \rightsquigarrow \langle \overrightarrow{\ell = \texttt{if } L \texttt{ then } M.\ell \texttt{ else } N.\ell} \rangle \quad (\text{if } M, N \text{ have type } \langle \overrightarrow{\ell : \sigma} \rangle)$$

B.2 Additional rule for NRC

For NRC, the following rewrite rule is added to the common rules:

$$\texttt{if } L \texttt{ then } M \texttt{ else } N \rightsquigarrow (M \texttt{ where } L) \cup (N \texttt{ where } \neg L) \quad (\text{if } M, N \text{ have type } \{\sigma\} \text{ and } N \neq \emptyset)$$

B.3 Additional rules for $\mathrm{NRC}_{\mathrm{opt}}$

For $\mathrm{NRC}_{\mathrm{opt}}$, the following rewrite rules are added to those of NRC:

$$\texttt{case none of } (\texttt{none} \Rightarrow N_1 \mid \texttt{some}(x) \Rightarrow N_2) \rightsquigarrow N_1$$

$$\texttt{case some}(M) \texttt{ of } (\texttt{none} \Rightarrow N_1 \mid \texttt{some}(x) \Rightarrow N_2) \rightsquigarrow N_2[M/x]$$

We believe that the case analysis rules, being a special case of sum types, are well behaved and preserve the strong normalization property.

B.4 Additional rules for $\mathrm{NRC}_{\mathrm{null}}$

For $\mathrm{NRC}_{\mathrm{null}}$, the following rewrite rules are added to the common rules:

$$M \texttt{ where null} \rightsquigarrow \emptyset$$

$$\begin{aligned}&\texttt{if } L \texttt{ then } M \texttt{ else } N\\ &\quad\rightsquigarrow (M \texttt{ where } L) \cup (N \texttt{ where } (\texttt{isNull}(L) \vee \neg L))\\ &\qquad\qquad (\text{if } M, N \text{ have type } \{\sigma\} \text{ and } N \neq \emptyset)\end{aligned}$$

$$\texttt{isNull}(\texttt{null}) \rightsquigarrow \texttt{true} \qquad \texttt{isNull}(c) \rightsquigarrow \texttt{false}$$

$$f(\ldots \texttt{null} \ldots) \rightsquigarrow \texttt{null} \qquad (M \texttt{ where null}) \rightsquigarrow \emptyset$$

Notice that the if-splitting rule is refined to account for the case where the condition is null; this additional check preserves Girard-Tait reducibility and we thus believe the rewrite system to be strongly normalizing.