Row-based Effect Types for Database Integration

Sam Lindley
The University of Edinburgh
Sam.Lindley@ed.ac.uk

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

Abstract

We present CORELINKS, a call-by-value variant of System F with row polymorphism, row-based effect types, and implicit subkinding, which forms the basis for the Links web programming language. We focus on extensions to CORELINKS for database programming. The effect types support abstraction over database queries, while ensuring that queries are translated predictably to idiomatic and efficient SQL at run-time. Subkinding statically enforces the constraint that queries must return a list of records of base type. Polymorphism over the presence of record labels supports abstraction over database queries, inserts, deletes and updates.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.3 [*Language Constructs and Features*]: Polymorphism

General Terms Languages, Theory

Keywords language integrated query, effect types, row types, polymorphism, normalisation

1. Introduction

Web programming is challenging because it involves coordinating two, three or more layers, each of which may be controlled by code written in a different programming language: for example a typical three-tier web application includes Java or Python code running on the server, HTML with embedded JavaScript on the client, and SQL for querying and updating data in a relational database.

Links [11] is a strict, statically typed, functional programming language for the web. Its main distinguishing feature is that it allows programmers to use the same programming language to write all three tiers of a web application: client, server and database. Links has many facets. This paper focuses on the effect type system of Links and its application to controlling database access.

1.1 Comprehensions

Links provides native support for list comprehensions, similar to those found in languages such as Haskell and Python. Furthermore, database queries in Links are expressed using comprehensions. The connection between comprehensions and query languages is well-established [5]. A comprehension in Links is written:

$$\mathsf{for}\,(x \leftarrow M)\,N$$

This iterates over the elements of the list expression M, binding them to x in the list expression N, and concatenating the results. In Haskell the above comprehension would be equivalent to $[z \mid x \leftarrow M, z \leftarrow N]$ or $concatMap\ (\lambda x \rightarrow N)\ M$. (The Links style *for comprehension* is more convenient than the classic Haskell style comprehension for our purposes, as it supports a cleaner rewriting theory. It is an instance of monadic let binding.)

1.2 A naive approach

The original implementation of Links [11] took a direct, somewhat ad hoc, approach to database integration. The strategy was simply to try to syntactically identify comprehensions in a source program that accessed database tables and translate them into SQL queries. For instance:

$$\begin{array}{l}\mathsf{for}\,(e \leftarrow \mathsf{asList}\ employees)\\ \quad[\langle name = e.name\rangle]\end{array}$$

would be translated to:

$$\begin{array}{l}\mathsf{select}\ e.name\ \mathsf{as}\ name\\ \mathsf{from}\ employees\ \mathsf{as}\ e\end{array}$$

(Lists are introduced with teletype square brackets `[]`, records are introduced with angle brackets $\langle\rangle$, and database tables are coerced to lists of records with asList. In order to reduce syntactic noise we deviate slightly from actual Links syntax.)

The direct approach works smoothly if the source comprehension includes only constructs that can be translated to SQL. If, however, the comprehension contains other features, such as general recursion, then it is much harder to perform an efficient translation. The direct approach would load entire database tables into main memory. Small changes to source code could lead to large and unpredictable changes in performance. For instance:

$$\begin{array}{l}\mathsf{for}\,(e \leftarrow \mathsf{asList}\ employees)\\ \quad[\langle name = reverse(e.name)\rangle]\end{array}$$

would be translated into a query that loads the entire *employees* table into memory along with a separate loop executed by the Links run-time to project out and reverse all the names (because the *reverse* operation is not available in SQL).

To make matters worse, it is difficult to see how to effectively extend the direct approach to support abstraction over queries and dynamic query generation. In a functional programming language the abstraction mechanism of choice is the first-class function, or closure. But SQL does not support closures, so in order to translate a comprehension containing closures to SQL one would have to attempt to inline the closures and then decide whether the resulting term could be translated to SQL. For instance:

$$\begin{array}{l}\mathsf{for}\,(e \leftarrow \mathsf{asList}\ employees)\\ \mathsf{where}\,(p(e))\\ \quad[\langle name = e.name\rangle]\end{array}$$

could be translated to a single query only if p could be inlined. Otherwise, the entire *employees* table would have to be loaded into memory before performing the filtering separately.

TLDI'12, January 28, 2012, Philadelphia, PA, USA.

1.3 An effective approach

The solution to all these problems adopted by Links 0.5 and later is to use effect types to restrict the behaviour of database code. Cooper [10] presents a strongly normalising rewrite system for a monomorphic nested relational calculus that illustrates the main idea. A single effect, called **wild** in Links, is assigned to any code that *cannot* be run on the database as part of a query. In particular, general recursion is wild, but lambda abstractions are not. We say that code is *tame* if it does not have the **wild** effect. Cooper's key result is that any tame expression with flat relation type can be translated to a single SQL query. For instance, suppose we define the following function:

$$\begin{array}{l} \mathit{filterEmployees} = \\ \quad \lambda p.\, \mathsf{query} \\ \quad\quad (\mathsf{for}\, (e \leftarrow \mathsf{asList}\ \mathit{employees}) \\ \quad\quad\ \mathsf{where}\, (p(e)) \\ \quad\quad\quad [\langle \mathit{name} = e.\mathit{name} \rangle]) \end{array}$$

A *query expression* query e asserts that e must be tame and must have flat relation type. Thus, the predicate p which maps employee records to booleans must not have the **wild** effect. We might invoke:

$$\mathit{filterEmployees}(\lambda e.\, (e.\mathit{salary} < 20000))$$

as the predicate $\lambda e.\, (e.\mathit{salary} < 20000)$ is tame, and this would be translated to the SQL query:

$$\begin{array}{l} \mathsf{select}\ e.\mathit{name} \\ \mathsf{from}\ \mathit{employees}\ \mathsf{as}\ e \\ \mathsf{where}\ e.\mathit{salary} < 20000 \end{array}$$

However, if we wrote:

$$\mathit{filterEmployees}(\lambda e.\, \mathit{primes}(e.\mathit{salary}) > 2000)$$

where $\mathit{primes}(n)$ computes the number of primes less than n, then this would result in a type error as the primes function is wild.

Give that queries are normalised before being executed as SQL, one may consider allowing general recursion inside queries. The reason we disallow it is that the normalisation procedure necessarily reduces under lambdas and inlines everything. We cannot fully inline the call to primes without a concrete value for $e.\mathit{salary}$, which is only available *after* the database has been queried.

Cooper's language is a monomorphic nested relational calculus. We extend his approach to a polymorphic language with support for database update operations.

1.4 Polymorphism

Supporting polymorphism is not quite as straightforward as one might expect because we need a way of abstracting over the result types of queries while maintaining the constraint that the rows returned by a query must be flat. Suppose we wish to generalise our filtering operation to operate on any table with a name field:

$$\begin{array}{l} \mathit{filterName} = \\ \quad \lambda t.\, \lambda p.\, \mathsf{query} \\ \quad\quad (\mathsf{for}\, (x \leftarrow \mathsf{asList}\ t) \\ \quad\quad\ \mathsf{where}\, (p(x)) \\ \quad\quad\quad [\langle \mathit{name} = x.\mathit{name} \rangle]) \end{array}$$

We might give $\mathit{filterName}$ the following polymorphic type:

$$\begin{array}{ll} & \mathit{Table}(\mathit{name} : \mathit{String}; \rho) \\ \to & (\langle \mathit{name} : \mathit{String}; \rho \rangle \to^{\mathbf{wild}^\circ} \mathit{Bool}) \\ \to^{\mathbf{wild}^\circ} & \mathit{List}\ \langle \mathit{name} : \mathit{String} \rangle \end{array}$$

where ρ is a polymorphic *row variable* ranging over fields other than name, $\mathit{Table}(R)$ is the type of tables with rows of type $\langle R \rangle$, and the annotation $\mathbf{wild}^\circ$ denotes the absence of the wild effect.

Now suppose we wish to generalise further by allowing any table at all in conjunction with a projection function f to apply to each row that matches the predicate:

$$\begin{array}{l} \mathit{filterTable} = \\ \quad \lambda t.\, \lambda p.\, \lambda f.\, \mathsf{query} \\ \quad\quad (\mathsf{for}\, (x \leftarrow \mathsf{asList}\ t) \\ \quad\quad\ \mathsf{where}\, (p(x)) \\ \quad\quad\quad [f(x)]) \end{array}$$

We might attempt to assign the following type to $\mathit{filterTable}$:

$$\begin{array}{ll} & \mathit{Table}(\rho) \\ \to & (\langle \rho \rangle \to^{\mathbf{wild}^\circ} \mathit{Bool}) \\ \to & (\langle \rho \rangle \to^{\mathbf{wild}^\circ} \langle \sigma \rangle) \\ \to^{\mathbf{wild}^\circ} & \mathit{List}\ \langle \sigma \rangle \end{array}$$

A problem with this type is that although it constrains the argument functions to be tame, it does not ensure that the result type is flat. In order to smoothly handle this scenario, we introduce a subkinding system. We define a kind of base types that is a subkind of ordinary types, and correspondingly a kind of base rows that is a subkind of ordinary rows. Thus we constrain σ in the type of $\mathit{filterTable}$ to range over base rows:

$$\begin{array}{ll} & \mathit{Table}(\rho) \\ \to & (\langle \rho \rangle \to^{\mathbf{wild}^\circ} \mathit{Bool}) \\ \to & (\langle \rho \rangle \to^{\mathbf{wild}^\circ} \langle \sigma :: \mathit{BaseRow} \rangle) \\ \to^{\mathbf{wild}^\circ} & \mathit{List}\ \langle \sigma :: \mathit{BaseRow} \rangle \end{array}$$

Cooper's type system incorporates a standard subsumption rule for effects, which allows tame expressions to be used in a wild context, for instance. We instead adopt an approach based on row-polymorphism. An effect type is a row of effects. This integrates well with the rest of Links, which also uses row polymorphism for record and variant types. Making the row polymorphism explicit, and also explicitly writing the quantifiers, the full CORELINKS type for $\mathit{filterTable}$ is:

$$\begin{array}{ll} \multicolumn{2}{l}{\forall \rho^{\mathit{BaseRow}_\emptyset}.\forall \sigma^{\mathit{BaseRow}_\emptyset}.\forall \varepsilon^{\mathit{Row}_{\{\mathbf{wild}\}}}.\forall \varepsilon_1^{\mathit{Row}_\emptyset}.\forall \varepsilon_2^{\mathit{Row}_\emptyset}.} \\ & \mathit{Table}(\rho) \\ \to^{\varepsilon_1} & (\langle \rho \rangle \to^{(\mathbf{wild}^\circ : \langle\rangle ; \varepsilon)} \mathit{Bool}) \\ \to^{\varepsilon_2} & (\langle \rho \rangle \to^{(\mathbf{wild}^\circ : \langle\rangle ; \varepsilon)} \langle \sigma \rangle) \\ \to^{(\mathbf{wild}^\circ : \langle\rangle ; \varepsilon)} & \mathit{List}\ \langle \sigma \rangle \end{array}$$

Each row kind is annotated with a set of labels which *cannot* appear in an instantiation of the row variable. Readers familiar with traditional effect polymorphism may be surprised to see the same row variable ε attached to the predicate, the projection function, and the result arrow. We discuss the reasons for this in Section 2.3. The arrow annotations ε_1 and ε_2 allow $\mathit{filterTable}$ to be partially applied to the first two arguments in arbitrary contexts.

(In the source language, and in examples, we often omit universal quantifiers, type annotations on effects: writing $\mathbf{wild}^\circ$ instead of $\mathbf{wild}^\circ : \langle\rangle$, and variables such as ε_1 and ε_2 that only occur once in a type. Note that the latter convention means that $\to$ is equivalent to $\to^\varepsilon$ for some fresh ε rather than $\to^\cdot$ where $\cdot$ denotes an empty row of effects.)

1.5 Updates

Let us now consider the implications of typing database updates. In the examples above we have parameterised table types by a *row type* which constrains the labels and types of the fields of the table. The type system we present in the rest of the paper allows us to refine table types in order to allow further constraints to be imposed on the table fields. Instead of a single row type, we parameterise table types by three separate row types: a *read* row, a *write* row, and a *needed* row.

Suppose our $employees$ table has fields $id : Int$, $name : String$, $salary : Int$, and $isManager : Bool$, where id is an auto-generated primary key and $isManager$ has a default value of false. We would assign the $employees$ table the type:

$$\begin{array}{l} Table \\ \quad ((id : Int; isManager : Bool; name : String; salary : Int), \\ \qquad\quad (isManager : Bool; name : String; salary : Int), \\ \qquad\qquad\quad (name : String; salary : Int)) \end{array}$$

All fields appear in the read row, as all fields must always be readable. The id field is read-only, so it does not appear in the write row. The needed row includes all fields that *need* to be present in a record when inserting it into the table. Clearly only writeable fields can appear in this row. Furthermore, fields with default values do not appear in this row. Thus, neither id nor $isManager$ appear in the needed row.

Another constraint we would ideally like to enforce on table types is that the fields in the read, write and needed rows should be consistent in the sense that whenever $\ell : A$ appears in one row and $\ell : B$ in another then $A = B$. It is straightforward to statically enforce consistency for concrete database tables. However, it is less straightforward to enforce consistency for polymorphic code that abstracts over operations on tables. Nevertheless, we make a step towards static enforcement by adopting a row type system in which it is possible to specify that a field must have a particular type *if* it is present, without committing to whether or not it is actually present. For instance, we can write the following function, which takes any table with an integer $salary$ field and a predicate, and doubles the $salary$ field of every row in the table satisfying the predicate:

$$\begin{array}{l} doubleSalaries = \\ \quad \lambda t.\, \lambda p.\, \mathsf{query} \\ \qquad \mathsf{update}\ (x \Leftarrow t) \\ \qquad \mathsf{where}\ (p(x)) \\ \qquad\quad \mathsf{set}\ \langle salary = salary * 2 \rangle \end{array}$$

The type of $doubleSalaries$ is:

$$\begin{array}{l} \forall \rho^{BaseRow_{\{salary\}}}.\forall \varepsilon^{Row_{\{\mathbf{wild}\}}}. \\ \qquad\quad Table((salary^{\bullet} : Int; \rho), \\ \qquad\qquad\quad (salary^{\bullet} : Int; \rho), \\ \qquad\qquad\quad (salary^{\theta} : Int; \rho)) \\ \to \qquad\quad (\langle \rho \rangle \to^{(\mathbf{wild}^{\circ};\varepsilon)} Bool) \\ \to^{(\mathbf{wild}^{\bullet};\varepsilon)} \langle \rangle \end{array}$$

This type exhibits *presence polymorphism*. The presence type variable θ can be instantiated to present ($\bullet$) or absent ($\circ$) according to whether or not the $salary$ field is present in the needed row of the table. If it is present then its type must be Int. We often omit present ($\bullet$) annotations.

The **wild** effect is absent from the predicate because the predicate must be translated to SQL. On the other hand, the **wild** effect is present on the final result arrow, as updates cannot appear inside database queries.

1.6 Concurrency

Another effect type provided by Links, **hear**, is used for providing statically-typed asynchronous message-passing concurrency inspired by Erlang [1]. The **hear** effect is parameterised by the type of messages that the current process is able to receive. The CORELINKS effect system evolved from an early implementation of concurrency for Links [11], in which arrow types were annotated with a mailbox type.

In Links, the concurrency design is particularly useful for managing user interfaces on the client-side of web applications. The implementation is described in detail elsewhere [9, 11].

In the current version of Links, concurrency is handled using the same effect typing mechanism as database integration. However, in this paper, we focus on database integration. We are currently formalising the semantics and type system for Links' Erlang-style concurrency, including a proof of type soundness.

1.7 Contributions and outline

The main contributions of this paper are as follows.

- A core language for effects based on System F extended with row polymorphism and presence polymorphism.
- A refinement of Cooper's approach to query normalisation whereby normalisation is decomposed into a standard reduction relation and a structurally recursive function, which makes proving termination considerably easier.
- An extension to support query normalisation for a polymorphic language including a subkinding system to smoothly support abstraction over flat relations.
- A design for accurately typing insert, delete and update operations taking advantage of presence polymorphism.
- Operational semantics for CORELINKS extended with database query and update support along with type soundness proofs.
- A proof of correctness of query evaluation.

The rest of the paper is structured as follows. Section 2 introduces the syntax and static semantics of CORELINKS. Section 3 describes an extension of CORELINKS with database operations. Section 4 gives the dynamic semantics for CORELINKS extended with database operations and proves it correct. Section 5 discusses the implementation of CORELINKS as part of the Links web programming language and Section 6 discusses related work. Section 7 concludes.

2. The core language

2.1 Syntax

Types and kinds The syntax of types and kinds is given below.

$$\begin{array}{lll} \text{Ordinary types} & A, B, C & ::= Int \mid Bool \mid String \\ & & \quad \mid A \to^{E} B \mid \langle R \rangle \mid [R] \\ & & \quad \mid \forall \alpha^{K}.A \mid List\ A \mid \alpha \\ \text{Row types} & R, S, E & ::= \cdot \mid \ell^{P} : A; R \mid \rho \\ \text{Presence types} & P & ::= \circ \mid \bullet \mid \theta \\ \\ \text{Types} & T & ::= A \mid R \mid P \\ \text{Type variables} & \alpha, \rho, \varepsilon, \theta & \\ \\ \text{Labels} & \ell & \\ \text{Label sets} & \mathcal{L} & ::= \{\ell_1, \ldots, \ell_k\} \\ \\ \text{Kinds} & K & ::= Type \mid BaseType \\ & & \quad \mid Row_{\mathcal{L}} \mid BaseRow_{\mathcal{L}} \\ & & \quad \mid Presence \end{array}$$

The base types are integers, booleans and strings. The function type $A \to^{E} B$ takes an argument of type A, returns a value of type B and has effects E. The record type $\langle R \rangle$ has fields given by the labels of row R. The variant type $[R]$ admits tagged values given by the labels of row R. The polymorphic type $\forall \alpha^{K}.A$ is parameterised over the type variable α of kind K. The type $List\ A$ is the type of lists whose elements have type A.

The full version of Links includes equi-recursive types, which allows us to use variant types as a structural alternative to the algebraic datatypes of languages such as ML and Haskell. Our variants are similar to OCaml's polymorphic variants. The main differences

are that we use explicit row variables, whereas OCaml uses constraints; we support negative presence information, whereas OCaml does not; and our case construct refines variant types, whereas OCaml's equivalent does not. Like OCaml, the full version of Links also supports subtyping (on records as well as variants) with an explicit upcast operator. As in OCaml, upcasts are occasionally useful. However, one of the attractions of row-typing is that it allows many programs that would otherwise require subtyping to be typed using polymorphism instead.

Effects The key effect we focus on in this paper is the **wild** effect. We say that a computation that *can* be run on the database is *tame* and a computation that *cannot* be run on the database is *wild*. In particular, the body of a query expression must be tame.

Rows Records, variants and effect are all defined in terms of rows. A row type includes a list of distinct labels, each of which is parameterised by a presence type and an ordinary type. The presence type indicates whether a label is present $\bullet$ or absent $\circ$. The ordinary type indicates the type of values associated with that label if it is present.

Row types are either *closed* or *open*. A closed row type ends in $\cdot$. An open row type ends in a *row variable* ρ. A term of closed row type can have only the labels explicitly listed in the type. The row variable in an open row type can be instantiated in order to extend the row type with additional labels. As usual, we identify rows up to reordering of labels.

$$\ell_1^{P_1} : A_1; \ell_2^{P_2} : A_2; R = \ell_2^{P_2} : A_2; \ell_1^{P_1} : A_1; R$$

Furthermore, absent labels in closed rows are redundant:

$$\ell^{\circ} : A; \ell_1^{P_1} : A_1, \ldots; \ell_n^{P_n} : A_n; \cdot = \ell_1^{P_1} : A_1, \ldots; \ell_n^{P_n} : A_n; \cdot$$

Presence types Unlike most other row type systems, but like Remy's $\mathbf{\Pi ML}'$ [22], the type of a label is independent of whether or not it is present. We make essential use of this feature in typing the database update operations. (It also allows us to implement the equivalent of OCaml's < constraints on polymorphic variants [12].)

Kinds Types in CORELINKS are classified into kinds. Ordinary types have kind Type. Row types R have kind $\mathit{Row}_{\mathcal{L}}$ where $\mathcal{L}$ is a set of labels not allowed in R. Presence types have kind $\mathit{Presence}$.

In addition to the three main kinds, CORELINKS supports a basic subkinding discipline. Base types have kind $\mathit{BaseType}$ which is a subkind of Type. Similarly, rows S restricted to base type have kind $\mathit{BaseRow}_{\mathcal{L}}$ where $\mathcal{L}$ is a set of labels not allowed in S, and this is a subkind of $\mathit{Row}_{\mathcal{L}}$. The subkinding discipline is used to enforce the constraint that database queries must return a list of records of base type (and similarly the constraint that values appearing in database inserts and updates must have base type). Potentially, it could be useful to add general support for user-defined subkinds, and some sort of overloading mechanism, but that is beyond the scope of this paper.

Terms The syntax of terms is given below.

$$\begin{aligned} L, M, N ::= {} & x \mid c \\ \mid {} & \lambda x^A.\, M \mid L\ M \\ \mid {} & \Lambda \alpha^K.\, M \mid M\ T \\ \mid {} & \langle\rangle \mid \langle \ell = M; N \rangle \mid M.\ell \\ \mid {} & (\ell\ M)^R \mid \mathsf{case}\ L\ \mathsf{of}\ \ell\ x \to M; y \to N \\ \mid {} & \mathsf{case}_{\perp}\ L \\ \mid {} & \mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N \\ \mid {} & [M] \mid \mathsf{for}\ (x \leftarrow M)\ N \\ \mid {} & []^A \mid M +\!\!+ N \\ \mid {} & \mathsf{rec}\ f^A\ x^{A'}.\, M \end{aligned}$$

We let x range over term variables and c range over constants. Term-level type abstractions $\Lambda\alpha^K.\, M$ are annotated with kinds. Records are introduced with the unit record $\langle\rangle$ and record extension $\langle \ell = M; N \rangle$ constructs. They are eliminated with projection $M.\ell$. Dually, variants are introduced with the injection $\ell\ M$ and eliminated using the case $\mathsf{case}\ L\ \mathsf{of}\ \ell\ x \to M; y \to N$ and empty case $\mathsf{case}_{\perp}\ L$ constructs. We build in conditionals $\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N$ because they are crucial to the generation of SQL. For the same reason we include term forms for introducing empty lists $[]^A$, singleton lists $[M]$ and concatenating two lists $M +\!\!+ N$, and for comprehensions for eliminating lists. We make a distinction between non-recursive functions $\lambda x^A.\, M$ and recursive functions $\mathsf{rec}\ f^A\ x^{A'}.\, M$ because the latter cannot in general be run on the database. For readability we often omit type and kind annotations.

Constants We assume a signature Σ mapping constants to their types. We let n range over integer constants, and s range over string constants. In addition, we assume that the constants c include at least the following.

$$\begin{aligned} \mathsf{true} &: \mathit{Bool} \\ \mathsf{false} &: \mathit{Bool} \\ (\neg) &: \forall\varepsilon.\mathit{Bool} \to^{\varepsilon} \mathit{Bool} \\ (\wedge) &: \forall\varepsilon.\langle \mathit{Bool}, \mathit{Bool}\rangle \to^{\varepsilon} \mathit{Bool} \\ (\vee) &: \forall\varepsilon.\langle \mathit{Bool}, \mathit{Bool}\rangle \to^{\varepsilon} \mathit{Bool} \\ (+) &: \forall\varepsilon.\langle \mathit{Int}, \mathit{Int}\rangle \to^{\varepsilon} \mathit{Int} \\ (-) &: \forall\varepsilon.\langle \mathit{Int}, \mathit{Int}\rangle \to^{\varepsilon} \mathit{Int} \\ (\times) &: \forall\varepsilon.\langle \mathit{Int}, \mathit{Int}\rangle \to^{\varepsilon} \mathit{Int} \\ (=) &: \forall\alpha^{\mathit{BaseType}}.\forall\varepsilon.\langle \alpha, \alpha\rangle \to^{\varepsilon} \mathit{Bool} \\ (<) &: \forall\alpha^{\mathit{BaseType}}.\forall\varepsilon.\langle \alpha, \alpha\rangle \to^{\varepsilon} \mathit{Bool} \\ (>) &: \forall\alpha^{\mathit{BaseType}}.\forall\varepsilon.\langle \alpha, \alpha\rangle \to^{\varepsilon} \mathit{Bool} \end{aligned}$$

Values CORELINKS is a call-by-value calculus. The syntax of values is standard.

$$\begin{aligned} U, V, W ::= {} & c \\ \mid {} & \Lambda\alpha.\, M \mid \lambda x.\, M \\ \mid {} & \langle\rangle \mid \langle \ell = V; W\rangle \mid \ell\ V \\ \mid {} & \mathsf{rec}\ f\ x.\, M \\ \mid {} & [V_1, \ldots, V_n] \end{aligned}$$

The notation $[V_1, \ldots, V_n]$ stands for the concatenation of singleton lists $[V_1], \ldots, [V_n]$. We interpret list values modulo identity $[] +\!\!+ V \equiv V \equiv V +\!\!+ []$ and associativity $(U +\!\!+ V) +\!\!+ W \equiv U +\!\!+ (V +\!\!+ W)$.

Notation We use the following abbreviations:

$$\begin{aligned} \ell : A &\equiv \ell^{\bullet} : A \\ \ell^P &\equiv \ell^P : \langle\rangle \\ \langle A_1, \ldots, A_k\rangle &\equiv \langle 1 : A_1; \ldots; k : A_k; \cdot\rangle \\ \overline{\ell} &\equiv \ell_1, \ldots, \ell_k \\ \overline{\ell^P : A} &\equiv \ell_1^{P_1} : A_1, \ldots, \ell_k^{P_k} : A_k \\ \overline{\ell^{\bullet} : A} &\equiv \ell_1^{\bullet} : A_1, \ldots, \ell_k^{\bullet} : A_k \\ \overline{\ell^{\circ} : A} &\equiv \ell_1^{\circ} : A_1, \ldots, \ell_k^{\circ} : A_k \end{aligned}$$

We interpret n-ary record and case extension at the type and term levels in the obvious way. For instance:

$$\overline{\ell^P : A}; R \equiv \ell_1^{P_1} : A_1; \ldots; \ell_n^{P_n} : A_n; R$$

2.2 Typing and kinding judgements

The typing rules are given in Figure 1. Type environments map term variables to types. Kind environments map type variables to kinds.

$$\begin{aligned} &\text{Type environments} \quad \Gamma ::= x_1 : A_1, \ldots, x_k : A_k \\ &\text{Kind environments} \quad \Delta ::= \alpha_1 :: K_1, \ldots, \alpha_k :: K_k \end{aligned}$$

The typing judgement $\Delta; \Gamma \vdash M : A\ !\ E$ says that in kind environment Δ and type environment Γ, the term M has type A and effects E. We assume that Γ, A and E are well-kinded with

$\boxed{\Delta; \Gamma \vdash M : A \,!\, E}$

$$\textsc{Var}\ \frac{}{\Delta; \Gamma, x : A \vdash x : A \,!\, E} \qquad \textsc{Const}\ \frac{\Sigma(c) = A}{\Delta; \Gamma \vdash c : A \,!\, E} \qquad \textsc{Lam}\ \frac{\Delta; \Gamma, x : A \vdash M : B \,!\, E}{\Delta; \Gamma \vdash \lambda x^A.\, M : A \to^E B \,!\, E'} \qquad \textsc{App}\ \frac{\Delta; \Gamma \vdash L : A \to^E B \,!\, E \qquad \Delta; \Gamma \vdash M : A \,!\, E}{\Delta; \Gamma \vdash L\ M : B \,!\, E}$$

$$\textsc{PolyLam}\ \frac{\Delta, \alpha :: K; \Gamma \vdash M : A \,!\, \mathbf{wild}^\circ; E \qquad \alpha \notin FTV(\Gamma, E)}{\Delta; \Gamma \vdash \Lambda\alpha^K.\, M : \forall\alpha^K.A \,!\, \mathbf{wild}^P; E} \qquad \textsc{PolyApp}\ \frac{\Delta; \Gamma \vdash M : \forall\alpha^K.A \,!\, E \qquad \Delta \vdash T :: K \qquad \alpha \notin FTV(E)}{\Delta; \Gamma \vdash M\ T : A[\alpha := T] \,!\, E}$$

$$\textsc{Unit}\ \frac{}{\Delta; \Gamma \vdash \langle\rangle : \langle\rangle \,!\, E} \qquad \textsc{Extend}\ \frac{\Delta; \Gamma \vdash M : A \,!\, E \qquad \Delta; \Gamma \vdash N : \langle \ell^\circ : A'; R\rangle \,!\, E}{\Delta; \Gamma \vdash \langle \ell = M; N\rangle : \langle \ell^\bullet : A; R\rangle \,!\, E} \qquad \textsc{Project}\ \frac{\Delta; \Gamma \vdash M : \langle \ell^\bullet : A; R\rangle \,!\, E}{\Delta; \Gamma \vdash M.\ell : A \,!\, E}$$

$$\textsc{Inject}\ \frac{\Delta; \Gamma \vdash M : A \,!\, E}{\Delta; \Gamma \vdash (\ell\ M)^R : [\ell^\bullet : A; R] \,!\, E} \qquad \textsc{Case}\ \frac{\Delta; \Gamma \vdash L : [\ell^\bullet : A; R] \,!\, E \qquad \Delta; \Gamma, x : A \vdash M : B \,!\, E \qquad \Delta; \Gamma, y : [\ell^\circ : A'; R] \vdash N : B \,!\, E}{\Delta; \Gamma \vdash \mathsf{case}\ L\ \mathsf{of}\ \ell\ x \to M; y \to N : B \,!\, E}$$

$$\textsc{CaseZero}\ \frac{\Delta; \Gamma \vdash L : [] \,!\, E}{\Delta; \Gamma \vdash \mathsf{case}_\bot\ L : A \,!\, E} \qquad \textsc{If}\ \frac{\Delta; \Gamma \vdash L : \mathit{Bool} \,!\, E \qquad \Delta; \Gamma \vdash M : A \,!\, E \qquad \Delta; \Gamma \vdash N : A \,!\, E}{\Delta; \Gamma \vdash \mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N : A \,!\, E} \qquad \textsc{Singleton}\ \frac{\Delta; \Gamma \vdash M : A \,!\, E}{\Delta; \Gamma \vdash [M] : [A] \,!\, E}$$

$$\textsc{For}\ \frac{\Delta; \Gamma \vdash M : \mathit{List}\ A \,!\, E \qquad \Delta; \Gamma, x : A \vdash N : \mathit{List}\ B \,!\, E}{\Delta; \Gamma \vdash \mathsf{for}\ (x \leftarrow M)\ N : \mathit{List}\ B \,!\, E} \qquad \textsc{Nil}\ \frac{}{\Delta; \Gamma \vdash []^A : [A] \,!\, E}$$

$$\textsc{Concat}\ \frac{\Delta; \Gamma \vdash M : [A] \,!\, E \qquad \Delta; \Gamma \vdash N : [A] \,!\, E}{\Delta; \Gamma \vdash M +\!\!+ N : [A] \,!\, E} \qquad \textsc{Rec}\ \frac{\Delta; \Gamma, f : A \to^{(\mathbf{wild}^P; E)} B, x : A \vdash M : B \,!\, \mathbf{wild}^P; E}{\Delta; \Gamma \vdash \mathsf{rec}\ f^{A \to^{(\mathbf{wild}^P; E)} B}\ x^A.\, M : A \to^{(\mathbf{wild}^\bullet; E)} B \,!\, E'}$$

Figure 1. Core typing rules

$$\textsc{Table}\ \frac{}{\Delta; \Gamma \vdash \mathsf{table}\ t : \Sigma(t) \,!\, E} \qquad \textsc{AsList}\ \frac{\Delta; \Gamma \vdash M : \mathit{Table}(S_r, S_w, S_n) \,!\, E}{\Delta; \Gamma \vdash \mathsf{asList}\ M : \mathit{List}\ \langle S_r\rangle \,!\, E} \qquad \textsc{Query}\ \frac{\Delta; \Gamma \vdash M : \mathit{List}\ \langle S\rangle \,!\, \mathbf{wild}^\circ; E \qquad \Delta \vdash S :: \mathit{BaseRow}_\mathcal{L}}{\Delta; \Gamma \vdash \mathsf{query}^S\ M : \mathit{List}\ \langle S\rangle \,!\, \mathbf{wild}^P; E}$$

$$\textsc{Insert}\ \frac{\begin{array}{l}\Delta; \Gamma \vdash L : \mathit{Table}(S_r, S_w, S_n) \,!\, \mathbf{wild}^P; E \\ \Delta; \Gamma \vdash M : \mathit{List}\ \langle \overline{\ell^\bullet : A}\rangle \,!\, \mathbf{wild}^{P'}; E \qquad S_r = (\overline{\ell^\bullet : A}; S) \qquad S_w = (\overline{\ell^\bullet : A}; S') \qquad S_n = (\overline{\ell^{P''} : A}; \cdot)\end{array}}{\Delta; \Gamma \vdash \mathsf{insert}\ L\ \mathsf{values}\ M : \langle\rangle \,!\, \mathbf{wild}^\bullet; E}$$

$$\textsc{Update}\ \frac{\begin{array}{l}\Delta; \Gamma \vdash L : \mathit{Table}(S_r, S_w, S_n) \,!\, \mathbf{wild}^P; E \qquad \Delta; \Gamma, x : \langle S_r\rangle \vdash M : \mathit{Bool} \,!\, \mathbf{wild}^\circ; E \\ \Delta; \Gamma, x : \langle S_r\rangle \vdash N : \langle \overline{\ell^\bullet : A}\rangle \,!\, \mathbf{wild}^\circ; E \qquad S_r = (\overline{\ell^\bullet : A}; S) \qquad S_w = (\overline{\ell^\bullet : A}; S') \qquad S_n = (\overline{\ell^{P'} : A}; S'')\end{array}}{\Delta; \Gamma \vdash \mathsf{update}\ (x \Leftarrow L)\ \mathsf{where}\ M\ \mathsf{set}^{\overline{\ell^\bullet : A}}\ N : \langle\rangle \,!\, \mathbf{wild}^\bullet; E}$$

$$\textsc{Delete}\ \frac{\Delta; \Gamma \vdash L : \mathit{Table}(S_r, S_w, S_n) \,!\, \mathbf{wild}^P; E \qquad \Delta; \Gamma, x : \langle S_r\rangle \vdash M : \mathit{Bool} \,!\, \mathbf{wild}^\circ; E}{\Delta; \Gamma \vdash \mathsf{delete}\ (x \Leftarrow L)\ \mathsf{where}\ M : \langle\rangle \,!\, \mathbf{wild}^\bullet; E}$$

Figure 2. Database typing rules

respect to Δ. If Δ, Γ, or E are empty, then they are sometimes omitted.

The typing rules are mostly straightforward; we only discuss the more interesting ones. The EXTEND rule is strict in the sense that it requires a label to be absent from a record before the record can be extended with the label. Dually, the CASE rule refines the type of the value being matched so that in the type of the variable bound by the default branch, the non-matched label is absent.

Aside The full version of Links also includes an operation to remove labels from a record, which allows one to define a record update operation that does not require the label to be absent. This can be used to define a non-strict record update operation that does not require the label being updated to be absent. Interestingly, if one works in CPS then this operation can be expressed in terms of an upcast. The dual operation on *first-class cases* [3] can be expressed directly with an upcast.

The only rules that actually change the effects are the ones for introducing term-level type abstractions and recursive functions. The POLYLAM rule is a relaxation of the ML value restriction. Rather than limiting the body of a Λ to a syntactic value, we insist that it is tame (observe that every syntactic value can always be assigned a tame effect). This ensures that impure extensions such as references or concurrency primitives cannot be used to break type soundness. The REC rule ensures that the body of a recursive function is wild, as recursive functions cannot be run on the database. (We do not attempt to generate stored procedures. Our goal is to abstract over a subset of SQL that does not include general recursion.)

Note that by design we do not support subsumption, and in particular we do not support sub-effecting, i.e., there is no implicit coercion from a computation with effects $\overline{\ell : A}; \cdot$ to one with effects $\overline{\ell : A}; R$. We discuss this design choice further in Section 2.3.

The kinding rules are given in Figure 3. The kinding judgement $\Delta \vdash A :: K$ says that in kind environment Δ, the type A has kind K. Type variables in the kind environment are well-kinded. The base types are Int, $Bool$ and $String$. The rules for forming function, record, variant, forall and list types follow the syntactic structure of types, as do the rules for forming presence types. Recall that $Row_{\mathcal{L}}$ is the kind of row types whose labels cannot appear in $\mathcal{L}$. (To be clear, this constraint applies equally to absent and present labels; it is a constraint on the form of *row types*. In contrast, ℓ° in a row type is a constraint on *terms*.) An empty row has kind $BaseRow_{\mathcal{L}}$ for any label set $\mathcal{L}$. The side-conditions $\ell \notin \mathcal{L}$ in EXTENDROW and EXTENDBASEROW ensure that row types have distinct labels. The restriction to base types in EXTENDBASEROWS allows only rows of base types to have kind $BaseRow_{\mathcal{L}}$. A row type can only be used to build a record, variant or function type if it has kind $Row_{\emptyset}$. This constraint ensures that any absent labels in an open row type must be mentioned explicitly. Subkinding is implicit: $BaseType$ is a subkind of $Type$ and $BaseRow_{\mathcal{L}}$ is a subkind of $Row_{\mathcal{L}}$. Implicit subkinding does not significantly affect type inference in the source language as there are no higher kinds and there is no polymorphism over kinds.

2.3 Effect rows vs standard effect polymorphism vs effect subtyping

Our treatment of effects is not dissimilar to standard effect polymorphism (for instance, see [24]). The key difference is that rather than using row types, standard effect polymorphism admits a union operation on effect types. In particular, it is possible to express an effect as the union of several effect variables.

$$E = \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$$

One way of characterising effect rows is as a restriction of standard effect polymorphism in which the union operation is only applicable if at least one of its arguments is monomorphic, i.e. any given effect type can include at most one effect variable. The advantage of imposing this restriction is that it puts an upper bound on the size of effect types: an effect row can only include those concrete effects that are relevant plus at most one effect variable.

In contrast, standard effect polymorphism can lead to effect types containing an unbounded number of polymorphic effect variables. For instance, each branch of an if or case can give rise to a new effect variable in the containing function. This blow-up in the size of types is not necessarily a problem for a compiler backend, but is undesirable for a source language. One of the goals of CORE-LINKS is to provide a target for *source* languages (primarily Links) that do support effect types and effect inference.

One way of reducing the size of effect types in standard effect polymorphism systems is to introduce subsumption into the type system (for instance, see [19]). That way a pure function can actually be assigned the empty effect rather than a polymorphic effect. The difficulty with subsumption is that it complicates the type inference algorithm, and can itself lead to large types arising from a proliferation of subtyping constraints.

3. Database programming

In order to support database access we add one type constructor and six new syntactic forms.

$$\text{Types} \quad A ::= \cdots \mid Table(S_r, S_w, S_n)$$

The type $Table(S_r, S_w, S_n)$ encodes three different views on a table as rows which capture constraints associated with the fields of the table. The *read* row S_r contains all the fields that can be read from the table. The *write* row S_w contains all the fields that can be written in the table. The *needed* row S_n contains all the fields that must be supplied when performing an insert operation (because they do not have default values).

We overload the type signature for constants Σ to define a fixed schema mapping table names to table types. In other words we assume a collection of table constants ranged over by t.

The terms are extended as follows.

$$\begin{aligned} L, M, N ::= {} & \ldots \\ & \mid \mathsf{table}\ t \mid \mathsf{asList}\ M \mid \mathsf{query}^S\ M \\ & \mid \mathsf{insert}\ L\ \mathsf{values}\ M \\ & \mid \mathsf{update}\ (x \Leftarrow L)\ \mathsf{where}\ M\ \mathsf{set}^S\ N \\ & \mid \mathsf{delete}\ (x \Leftarrow L)\ \mathsf{where}\ M \end{aligned}$$

The database typing rules are given in Figure 2. This is where we really start to make use of the novel features of the type system.

A table handle $\mathsf{table}\ t$ represents the table t, whose type is $\Sigma(t)$. The coercion $\mathsf{asList}\ M$ reads an entire table M from the database as a list of records. A query expression $\mathsf{query}^S\ M$ statically guarantees that M will be translated to at most one SQL query at run-time, and the results will have flat relation type $\langle S \rangle$. Despite the fact that the only primitive for reading from the database is asList, the normalisation algorithm will in fact ensure that only the necessary data is extracted. For instance, the example in Section 1.3 of filtering by employees salary only returns the names of the employees.

The insert, update and delete commands translate directly to their SQL counterparts. The command $\mathsf{insert}\ L\ \mathsf{values}\ M$ inserts the list of records M into the table L. The fields of the elements of M must include all the needed fields of L. The command $\mathsf{update}\ (x \Leftarrow L)\ \mathsf{where}\ M\ \mathsf{set}^S\ N$ updates each row in the table L with N whenever M is true. The variable x is bound in N and M to the contents of the current row. The fields of N can be any writeable subset of the fields of L. The command $\mathsf{delete}\ (x \Leftarrow L)\ \mathsf{where}\ M$

$\boxed{\Delta \vdash T :: K}$

TyVar
$$\frac{}{\Delta, \alpha :: K \vdash \alpha :: K}$$

Int
$$\frac{}{\Delta \vdash Int :: BaseType}$$

Bool
$$\frac{}{\Delta \vdash Bool :: BaseType}$$

String
$$\frac{}{\Delta \vdash String :: BaseType}$$

Function
$$\frac{\Delta \vdash A :: Type \qquad \Delta \vdash E :: Row_{\emptyset} \qquad \Delta \vdash B :: Type}{\Delta \vdash A \to^{E} B :: Type}$$

Record
$$\frac{\Delta \vdash R :: Row_{\emptyset}}{\Delta \vdash \langle R \rangle :: Type}$$

Variant
$$\frac{\Delta \vdash R :: Row_{\emptyset}}{\Delta \vdash [R] :: Type}$$

ForAll
$$\frac{\Delta, \alpha :: K \vdash A :: Type}{\Delta \vdash (\forall \alpha^{K}.A) :: Type}$$

List
$$\frac{\Delta \vdash A :: Type}{\Delta \vdash List\ A :: Type}$$

Present
$$\frac{}{\Delta \vdash \circ :: Presence}$$

Absent
$$\frac{}{\Delta \vdash \bullet :: Presence}$$

EmptyBaseRow
$$\frac{}{\Delta \vdash \cdot :: BaseRow_{\mathcal{L}}}$$

ExtendRow
$$\frac{\ell \notin \mathcal{L} \quad \Delta \vdash R :: Row_{\mathcal{L} \cup \{\ell\}} \quad \Delta \vdash A :: Type \quad \Delta \vdash P :: Presence}{\Delta \vdash (\ell^{P} : A; R) :: Row_{\mathcal{L}}}$$

ExtendBaseRow
$$\frac{\ell \notin \mathcal{L} \quad \Delta \vdash S :: BaseRow_{\mathcal{L} \cup \{\ell\}} \quad \Delta \vdash B :: BaseType \quad \Delta \vdash P :: Presence}{\Delta \vdash (\ell^{P} : B; S) :: BaseRow_{\mathcal{L}}}$$

UpcastBaseType
$$\frac{\Delta \vdash B :: BaseType}{\Delta \vdash B :: Type}$$

UpcastBaseRow
$$\frac{\Delta \vdash S :: BaseRow_{\mathcal{L}}}{\Delta \vdash S :: Row_{\mathcal{L}}}$$

Figure 3. Kinding rules

deletes each row from the table L for which M is true. The variable x is bound in M to the contents of the current row.

Table handles are the only new value form.

$$V, W ::= \cdots \mid \mathsf{table}\ t$$

The Query rule enforces the constraints that the body of a query expression must be tame and return a list of records of base type. The Insert rule ensures that: a) any fields being written are compatible with the writeable fields of the table, and b) all needed fields are included in the values being written. Similarly, the Update rule ensures that any fields being updated are compatible with the writeable fields of the table. It also ensures that the where and set clauses are tame allowing all updates to be translated directly to SQL. Similarly, the Delete rule ensures that the where clause is tame. We have chosen to enforce the constraint that all writeable fields are also readable, but it would be perfectly possible to relax this constraint. In the Insert and Update rules, the types of the labels in the needed row are constrained to be consistent with those of the other rows. The presence information is, on the other hand, unconstrained as we can infer nothing about it solely from an insert or update operation. In the Insert rule the needed row is closed

β-rules

$$\begin{aligned}
(\lambda x.\, M)\ V &\longrightarrow M[x := V] \\
(\mathsf{rec}\ f\ x.\, M)\ V &\longrightarrow M[f := \mathsf{rec}\ f\ x.\, M,\ x := V] \\
(\Lambda \alpha.\, M)\ T &\longrightarrow M[\alpha := T] \\
\langle \ell = V; W \rangle.\ell &\longrightarrow V \\
\langle \ell = V; W \rangle.\ell' &\longrightarrow W.\ell',\ \mathsf{if}\ \ell \neq \ell' \\
\mathsf{case}\ \ell\ V\ \mathsf{of}\ \ell\ x \to M; z \to N &\longrightarrow M[x := V] \\
\mathsf{case}\ \ell\ V\ \mathsf{of}\ \ell'\ x \to M; z \to N &\longrightarrow N[z := \ell\ V],\ \mathsf{if}\ \ell \neq \ell' \\
\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ M\ \mathsf{else}\ N &\longrightarrow M \\
\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ M\ \mathsf{else}\ N &\longrightarrow N \\
\mathsf{for}\ (x \leftarrow [V])\ M &\longrightarrow M[x := V]
\end{aligned}$$

List deconstruction

$$\begin{aligned}
\mathsf{for}\ (x \leftarrow [])\ N &\longrightarrow [] \\
\mathsf{for}\ (x \leftarrow V +\!\!+ W)\ N &\longrightarrow (\mathsf{for}\ (x \leftarrow V)\ N) \\
&\quad +\!\!+ (\mathsf{for}\ (x \leftarrow W)\ N)
\end{aligned}$$

Evaluation contexts

$$\frac{M \longrightarrow M'}{\mathcal{E}[M] \longrightarrow \mathcal{E}[M']}$$

$$\begin{aligned}
\mathcal{E} ::= {} & [\,] \mid \mathcal{E}\ M \mid V\ \mathcal{E} \mid \mathcal{E}\ T \\
\mid {} & \langle \ell = \mathcal{E}; N \rangle \mid \langle \ell = V; \mathcal{E} \rangle \mid \mathcal{E}.\ell \\
\mid {} & \ell\ \mathcal{E} \mid \mathsf{case}\ \mathcal{E}\ \mathsf{of}\ \ell\ x \to M; y \to N \mid \mathsf{case}_{\perp}\ \mathcal{E} \\
\mid {} & \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ M\ \mathsf{else}\ N \\
\mid {} & [\mathcal{E}] \mid \mathcal{E} +\!\!+ M \mid V +\!\!+ \mathcal{E} \\
\mid {} & \mathsf{for}\ (x \leftarrow \mathcal{E})\ N
\end{aligned}$$

Figure 4. Operational semantics

because all needed fields must occur in the type of M, whereas in the Update rule the needed row is open because there is no lower bound on the number of fields that an update can modify.

4. Dynamic semantics

We start with a small-step semantics for CoreLinks (Figure 4) without the database operations. The statement $M \longrightarrow M'$ can be read as: term M evaluates to term M' in one step. The rules are all completely standard call-by-value β-reductions, except for the last two which are used to deconstruct lists.

Now we consider how to generate an SQL query. The goal is to convert the code inside a query expression into SQL. We will show that it is always possible to rewrite a tame term of flat relation type into the following normal form:

$$\begin{array}{lll}
\text{Queries} & L & ::= (+\!\!+)\ \overline{C} \\
\text{Comprehensions} & C & ::= \mathsf{for}\ (\overline{G})\ \mathsf{where}\ X\ [R] \\
\text{Generators} & G & ::= x \leftarrow t \\
\text{Records} & R & ::= \langle \overline{\ell = X} \rangle \\
\text{Base expressions} & X, Y, Z & ::= x.\ell \mid c(\overline{X}) \\
 & & \quad \mid \mathsf{if}\ X\ \mathsf{then}\ Y\ \mathsf{else}\ Z
\end{array}$$

where we use the following abbreviations:

$$\begin{aligned}
(+\!\!+)\ \overline{C} &\equiv C_1 +\!\!+ \ldots +\!\!+ C_n \\
\mathsf{for}\ (\overline{G})\ M &\equiv \mathsf{for}\ (G_1)\ \ldots\ \mathsf{for}\ (G_n)\ M \\
\mathsf{where}\ M\ N &\equiv \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ [] \\
t &\equiv \mathsf{asList}\ (\mathsf{table}\ t)
\end{aligned}$$

We assume that the order in which SQL queries return rows is non-deterministic. We discuss briefly how Links supports ordered SQL output in Section 5. Ignoring list ordering, normalised queries

β-rules

$$
\begin{array}{rcl}
(\lambda x.\, N)\ M & \rightsquigarrow & N[x := M] \\
(\Lambda \alpha.\, M)\ T & \rightsquigarrow & M[\alpha := T] \\
\langle \ell = M; N \rangle.\ell & \rightsquigarrow & M \\
\langle \ell = M; N \rangle.\ell' & \rightsquigarrow & N.\ell', \text{ if } \ell \neq \ell' \\
\mathsf{case}\ \ell\ L\ \mathsf{of}\ \ell\ x \to M; z \to N & \rightsquigarrow & M[x := L] \\
\mathsf{case}\ \ell\ L\ \mathsf{of}\ \ell'\ x \to M; z \to N & \rightsquigarrow & N[z := \ell\ L], \text{ if } \ell \neq \ell' \\
\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ M\ \mathsf{else}\ N & \rightsquigarrow & M \\
\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ M\ \mathsf{else}\ N & \rightsquigarrow & N \\
\mathsf{for}\ (x \leftarrow \mathtt{[}M\mathtt{]})\ N & \rightsquigarrow & N[x := M]
\end{array}
$$

List deconstruction

$$
\begin{array}{rcl}
\mathsf{for}\ (x \leftarrow \mathtt{[]})\ M & \rightsquigarrow & \mathtt{[]} \\
\mathsf{for}\ (x \leftarrow M_1 +\!\!+ M_2)\ N & \rightsquigarrow & (\mathsf{for}\ (x \leftarrow M_1)\ N) \\
 & & +\!\!+\ (\mathsf{for}\ (x \leftarrow M_2)\ N)
\end{array}
$$

Commuting conversions

$$
\begin{array}{rcl}
\mathcal{F}[\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N] & \rightsquigarrow & \mathsf{if}\ L\ \mathsf{then}\ \mathcal{F}[M]\ \mathsf{else}\ \mathcal{F}[N] \\
\mathsf{for}\ (x \leftarrow \mathsf{for}\ (y \leftarrow L)\ M)\ N & \rightsquigarrow & \mathsf{for}\ (y \leftarrow L)\ \mathsf{for}\ (x \leftarrow M)\ N
\end{array}
$$

Elimination frames

$$
\begin{array}{rcl}
\mathcal{F} ::= [\,]\ M \mid [\,]\ T \mid [\,].\ell \\
\quad \mid\ \mathsf{case}\ [\,]\ \mathsf{of}\ \ell\ x \to M; z \to N \\
\quad \mid\ \mathsf{if}\ [\,]\ \mathsf{then}\ M\ \mathsf{else}\ N \\
\quad \mid\ \mathsf{for}\ (x \leftarrow [\,])\ N
\end{array}
$$

Compatible closure

$$\frac{M \rightsquigarrow M'}{\mathcal{K}[M] \rightsquigarrow \mathcal{K}[M']}$$

where $\mathcal{K}$ ranges over one-hole contexts

Figure 5. Query rewriting

can be translated to SQL trivially, since the syntax of normalised queries is isomorphic to the following fragment of SQL:

$$
\begin{array}{lll}
\text{Queries} & L, M, N & ::= (\mathsf{union\ all})\ \overline{C} \\
\text{Comprehensions} & C & ::= \mathsf{select}\ R\ \mathsf{from}\ \overline{G}\ \mathsf{where}\ X \\
\text{Generators} & G & ::= t\ \mathsf{as}\ x \\
\text{Record terms} & R & ::= \overline{X\ \mathsf{as}\ \ell} \\
\text{Base terms} & X, Y, Z & ::= x.\ell \mid c(\overline{X}) \\
 & & \quad \mid\ \mathsf{case\ when}\ X\ \mathsf{then}\ Y\ \mathsf{else}\ Z\ \mathsf{end}
\end{array}
$$

Strictly speaking, this is not quite a subset of SQL as SQL cannot handle comprehensions in which R is a row with no fields, or empty unions. We ignore these idiosyncrasies.

4.1 Normalisation

Query evaluation relies on the query normalisation function $norm$ defined in Figures 5 and 6. This is invoked whenever evaluation needs to access the database. Cooper [10] defines a similar normalisation algorithm for a monomorphic language. His termination proof is somewhat intricate because some of the rewrite rules do not fit standard patterns. In order to avoid such difficulties in our more challenging polymorphic setting, we have decomposed the normalisation algorithm into two stages. The first stage performs standard symbolic evaluation, that is β-reductions, list deconstruction, and commuting conversions, defined through a reduction relation $\rightsquigarrow$, thus it is amenable to standard termination arguments. The second stage takes the output of the first stage as input. It is defined as a structurally recursive function that returns a term in the desired normal form.

Each β rule in the $\rightsquigarrow$ relation arises from a corresponding β rule in the $\longrightarrow$ relation (and similarly for the list deconstruction rules). Notice however, that because we are only interested in applying query normalisation rules to tame terms, the β rules are the more general call-by-name variants. Furthermore, the rules are closed under all contexts rather than just evaluation contexts. This means that it is possible to perform some reductions that appear unsound: although the whole query term must be tame, it may refer to functions whose bodies are wild, providing those functions are not actually used. For instance, suppose that we add a `print` command that outputs a string and whose effect is $\mathbf{wild}$. Now the term:

$$\mathsf{query}\ (\lambda x.\ \mathtt{[]})\ ((\lambda y.\, \lambda z.\, \langle y, y \rangle)(\mathsf{print}\ \mathtt{"foo"}))$$

rewrites to:

$$\mathsf{query}\ (\lambda x.\ \mathtt{[]})\ ((\lambda z.\, \langle \mathsf{print}\ \mathtt{"foo"}, \mathsf{print}\ \mathtt{"foo"} \rangle))$$

which appears to duplicate the `print` command. However, this then rewrites to:

$$\mathsf{query}\ \mathtt{[]}$$

The purpose of the commuting conversions is to expose further β-reductions. They are necessary because not all values are available at normalisation time. (It is not necessary to perform commuting conversions for hoisting cases out of elimination contexts because cases can always be eliminated from closed flat relational queries by β-reduction alone.)

The syntax of $\rightsquigarrow$-normal forms is given by the following grammar:

$$
\begin{array}{lll}
\text{Terms} & M, N & ::= \mathtt{[}R\mathtt{]} \mid \mathsf{for}\ (x \leftarrow t)\ M \mid t \\
 & & \quad \mid\ \mathtt{[]} \mid M +\!\!+ N \\
 & & \quad \mid\ \mathsf{if}\ X\ \mathsf{then}\ M\ \mathsf{else}\ N \\
\text{Records} & R, S & ::= x \mid \overline{\langle \ell = X \rangle} \\
 & & \quad \mid\ \mathsf{if}\ X\ \mathsf{then}\ R\ \mathsf{else}\ S \\
\text{Base expressions} & X, Y, Z & ::= x.l \mid c(\overline{X}) \\
 & & \quad \mid\ \mathsf{if}\ X\ \mathsf{then}\ Y\ \mathsf{else}\ Z
\end{array}
$$

Given a $\rightsquigarrow$-normal form M, the second stage of normalisation computes $\|M\|_{[],\mathsf{true}}$, which splits M into a list of comprehensions, which when concatenated together gives the desired normal form. It is defined with respect to the labels $\overline{\ell}$ of the output relation. (In order to distinguish the meta language from the object languages, we write roman square brackets $[-]$ for the meta level list constructor and $\oplus$ for the meta level concatenation of two lists — as opposed to teletype square brackets `[]` and $+\!+$ for object languages.)

The function $\|-\|_{\overline{G},X}$ splits its argument aggregating the currently active generators $\overline{G}$ and the current `where` clause X as it proceeds. The auxiliary $\|-\|$ function recurses on the record at the tail of a comprehension. The auxiliary $\|-\|_i$ function computes the base expression associated with the ℓ_i component of the record. Together $\|-\|$ and $\|-\|_i$ perform any necessary η-expansion for record variables, and push conditionals inside records. These operations are necessary as SQL supports neither record variables nor conditionals over records.

Given a tame flat relational term $M : \mathtt{[}\overline{\langle \ell : A \rangle}\mathtt{]}$, we write $norm_{\overline{\ell}}(M)$ for the function that first applies $\rightsquigarrow$-normalisation and then splits the result to obtain a normalised query. It is well-defined because $\rightsquigarrow$ is confluent.

4.2 The database operations

Now we extend the small-step rules to include the database operations (Figure 7). We model a database DB as a record of tables. Each table t is a *bag* (or *multiset*) of flat records of type $\Sigma(t)$. We write $\Sigma \vdash DB$ to assert that the value of each table t of DB satisfies $\vdash DB.t : \Sigma(t)$. The small-step evaluation relation is extended to keep track of the database. The statement $DB; M \longrightarrow M'; DB'$

$$
\begin{array}{rcl}
norm_{\overline{\ell}}(M) & = & (+\!\!+)\,(\|N\|_{[],\mathsf{true}}) \\
normRecord_{\overline{\ell}}(M) & = & \|N\| \\
normBase(M) & = & N
\end{array}
$$

where $M \rightsquigarrow^* N \not\rightsquigarrow$ and:

$$
\begin{array}{rcl}
\|\,[R]\,\|_{\overline{G},X} & = & [\mathsf{for}\,(\overline{G})\,\mathsf{where}\,X\;[\|R\|]] \\
\|\mathsf{for}\,(x \leftarrow t)\,M\|_{\overline{G},X} & = & \|M\|_{\overline{G}\oplus[x\leftarrow t],X} \\
\|t\|_{\overline{G},X} & = & \|\,[x]\,\|_{\overline{G}\oplus[x\leftarrow t],X}, \quad x \text{ fresh} \\
\|\,[]\,\|_{\overline{G},X} & = & [] \\
\|M +\!\!+ N\|_{\overline{G},X} & = & \|M\|_{\overline{G},X} \oplus \|N\|_{\overline{G},X} \\
\|\mathsf{if}\,Y\,\mathsf{then}\,M\,\mathsf{else}\,N\|_{\overline{G},X} & = & \|M\|_{\overline{G},X\wedge Y} \oplus \|N\|_{\overline{G},X\wedge\neg Y}
\end{array}
$$

$$
\|R\| = \langle \ell_1 = \|R\|_1, \ldots, \ell_n = \|R\|_n \rangle
$$

$$
\begin{array}{rcl}
\|x\|_i & = & x.\ell_i \\
\|\langle \ell_1 = X_1, \ldots, \ell_n = X_n\rangle\|_i & = & X_i \\
\|\mathsf{if}\,X\,\mathsf{then}\,R\,\mathsf{else}\,S\|_i & = & \mathsf{if}\,X\,\mathsf{then}\,\|R\|_i\,\mathsf{else}\,\|S\|_i
\end{array}
$$

Figure 6. Query normalisation

can be read as: given input database DB the term M evaluates to the term M' and produces the output database DB' in one step.

Evaluating $\mathsf{asList}\ (\mathsf{table}\,t)$ reads the entire table t and converts it to a list. In SQL this amounts to $\mathsf{select}\ *\ \mathsf{from}\,t$. In order to model the default behaviour of SQL, whereby no guarantee is made as to the order in which rows are returned, the $asList$ function takes a bag and non-deterministically converts it to a corresponding list. Evaluating a query expression $\mathsf{query}^S\ M$ first normalises $unQuery(M)$, and then runs the normalised term. The $unQuery$ function maps each sub-term of the form $\mathsf{query}\,N$ to N (thus normalisation ignores nested query constructors). We write $V \simeq W$ to mean that the list value V is a permutation of the list value W. Hence executing a query is non-deterministic as in SQL

Updates on the database are performed by the $update_{DB}$ function, which we leave abstract as our focus is the semantic properties of queries. The update operation performs query normalisation on the where and set clauses. Observe that evaluating the latter depends on knowing the labels that are being updated. The delete operation also normalises its where clause.

4.3 Type soundness

Preservation and progress properties hold for the operational semantics.

Proposition 1 (Preservation)**.** *If $\Delta;\Gamma \vdash M : A\,!\,E$, $\Sigma \vdash DB$ and $DB; M \longrightarrow M'; DB'$ then $\Delta;\Gamma \vdash M' : A\,!\,E$ and $\Sigma \vdash DB'$.*

Proposition 2 (Progress)**.** *If $\Delta;\Gamma \vdash M : A\,!\,E$, then either M is a value or there exists a term M' and a database DB', such that $DB; M \longrightarrow M'; DB'$.*

As usual, preservation is proved by induction over the evaluation relation $\longrightarrow$ using a suitable substitution lemma. The only interesting cases are those rules that involve the $norm$ function. Preservation depends on a preservation property for this function, that is, if $\Delta;\Gamma \vdash M : [\langle \overline{\ell : A}\rangle]\,!\,\mathbf{wild}^\circ; E$ then $\Delta;\Gamma \vdash norm_{\overline{\ell}}(M) : [\langle \overline{\ell : A}\rangle]\,!\,\mathbf{wild}^\circ; E$. This in turn depends on a notion of preservation, or *subject reduction* for the $\rightsquigarrow$ relation.

Proposition 3 (Subject reduction)**.**

1. *If $\Delta;\Gamma \vdash M : A\,!\,E$ and $M \rightsquigarrow M'$, then there exists some E', such that $\Delta;\Gamma \vdash M' : A\,!\,E'$.*
2. *If $\Delta;\Gamma \vdash M : A!(\mathbf{wild}^\circ; E)$, then either $M \not\rightsquigarrow$, or there exists some M' such that $M \rightsquigarrow M'$ and $\Delta;\Gamma \vdash M' : A!(\mathbf{wild}^\circ; E)$.*

$\boxed{DB; M \longrightarrow N; DB'}$

$$
\frac{V = asList(DB.t)}{DB; \mathsf{asList}\ (\mathsf{table}\,t) \longrightarrow V; DB}
$$

$$
\frac{N = norm_{\overline{\ell}}(unQuery(M)) \qquad N \longrightarrow V \qquad V \simeq W}{DB; \mathsf{query}^{\overline{\ell:A}}\ M \longrightarrow W; DB}
$$

$$
\frac{DB' = update_{DB}(\mathsf{insert}\,(\mathsf{table}\,t)\,\mathsf{values}\,[\overline{R}])}{DB; \mathsf{insert}\,(\mathsf{table}\,t)\,\mathsf{values}\,[\overline{R}] \longrightarrow \langle\rangle; DB'}
$$

$$
\frac{X = normBase(M) \qquad R = normRecord_{\overline{\ell}}(N) \qquad DB' = update_{DB}(\mathsf{update}\,(x \Leftarrow \mathsf{table}\,t)\,\mathsf{where}\,X\,\mathsf{set}^{\overline{\ell:A}}\,R)}{DB; \mathsf{update}\,(x \Leftarrow \mathsf{table}\,t)\,\mathsf{where}\,M\,\mathsf{set}^{\ell:A}\,N \longrightarrow \langle\rangle; DB'}
$$

$$
\frac{X = normBase(M) \qquad DB' = update_{DB}(\mathsf{delete}\,(x \Leftarrow \mathsf{table}\,t)\,\mathsf{where}\,X)}{DB; \mathsf{delete}\,(x \Leftarrow \mathsf{table}\,t)\,\mathsf{where}\,M \longrightarrow \langle\rangle; DB'}
$$

Evaluation contexts

$$
\begin{array}{rcl}
\mathcal{E} & ::= & \ldots \\
& \mid & \mathsf{asList}\ \mathcal{E} \\
& \mid & \mathsf{insert}\ \mathcal{E}\ \mathsf{values}\ M \\
& \mid & \mathsf{insert}\ V\ \mathsf{values}\ \mathcal{E} \\
& \mid & \mathsf{update}\,(x \Leftarrow \mathcal{E})\,\mathsf{where}\,M\,\mathsf{set}^{\overline{\ell:A}}\,N \\
& \mid & \mathsf{delete}\,(x \Leftarrow \mathcal{E})\,\mathsf{where}\,M
\end{array}
$$

Figure 7. Operational semantics for the database operations

The first property is a weak form of subject reduction, which allows the effect type to vary. This is necessary because we are allowing call-by-name reduction. The second property states that whenever a reduction is possible, we can always choose one that does preserve effects. This can be achieved by adopting a reduction strategy that only contracts tame redexes. We can be sure that any impure redexes will eventually disappear as M itself is tame.

As usual, progress is proved by induction over typing derivations using a suitable substitution lemma. Again, the only interesting cases arise when rules involving the $norm$ function apply. Progress depends on this function terminating, which in turn depends on termination of the $\rightsquigarrow$ relation.

Theorem 4. *If $\Delta;\Gamma \vdash M : A\,!\,\mathbf{wild}^\circ; E$, then M is strongly normalising with respect to $\rightsquigarrow$.*

Proof sketch. The proof is by translation to F_ω extended with products, sums, unit and empty type including the standard commuting conversions as well as β-rules. We call this language $\mathrm{F}_\omega^{\times+10}$. Matthes [18] proves strong normalisation for a similar extension to System F and his proof scales up to $\mathrm{F}_\omega^{\times+10}$ (as well as other features such as positive recursive types).

We observe that for any given term, polymorphic records and variants can be straightforwardly simulated by products and sums. Let n be the number of distinct labels appearing in the term. The idea is to simulate each record by an n-ary product and each variant by an n-ary sum. We write $|\ell|$ for the position in a product encoding a record containing ℓ or a sum encoding a variant containing ℓ. Absent labels in a record are given the unit type, and absent labels in a variant type are given the zero type. Row variables are represented as $2n$ standard type variables: two for each label. Two are

$$
\begin{array}{rcl}
[\![Int]\!]_\sigma &=& Int \\
[\![String]\!]_\sigma &=& String \\
[\![Bool]\!]_\sigma &=& 1+1 \\
[\![A \to^E B]\!]_\sigma &=& [\![A]\!]_\sigma \to [\![B]\!]_\sigma \\
[\![\langle R \rangle]\!]_\sigma &=& (\times)\,[\![R]\!]_{\sigma,[1,^n,1],1} \\
[\![[R]]\!]_\sigma &=& (+)\,[\![R]\!]_{\sigma,[0,^n,0],0} \\
[\![\forall \alpha^{Type}.A]\!]_\sigma &=& \forall \alpha^\star.[\![A]\!]_\sigma = [\![\forall \alpha^{BaseType}.A]\!]_\sigma \\
[\![\forall \rho^{Row\,\mathcal{L}}.A]\!]_\sigma &=& \forall (\rho^\dagger).[\![A]\!]_\sigma = [\![\forall \rho^{BaseRow\,\mathcal{L}}.A]\!]_\sigma \\
[\![\forall \theta^{Presence}]\!]_\sigma &=& \forall \theta^{\star\to\star\to\star}.\theta\,[\![A]\!]_{\sigma[\theta\mapsto\bullet]}\,[\![A]\!]_{\sigma[\theta\mapsto\circ]} \\
[\![List\ A]\!]_\sigma &=& 1 + [\![A]\!]_\sigma \\
[\![\alpha]\!]_\sigma &=& \alpha
\end{array}
$$

$$
\begin{array}{rcl}
[\![\ell^\bullet : A; R]\!]_{\sigma,\overline{B},C} &=& [\![R]\!]_{\sigma,\overline{B}[[\ell]\mapsto[\![A]\!]_\sigma],C} \\
[\![\ell^\circ : A; R]\!]_{\sigma,\overline{B},C} &=& [\![R]\!]_{\sigma,\overline{B},C} \\
[\![\ell^\theta : A; R]\!]_{\sigma,\overline{B},C} &=& [\![R]\!]_{\sigma,\overline{B}[[\ell]\mapsto A'],C}, \\
& & \quad \text{where } A' = \begin{cases} [\![A]\!]_\sigma, & \text{if } \sigma(\theta) = \bullet \\ C, & \text{if } \sigma(\theta) = \circ \end{cases} \\
[\![\cdot]\!]_{\sigma,\overline{B},C} &=& \overline{B} \\
[\![\rho :: Row_{\mathcal{L}}]\!]_{\sigma,\overline{B},C} &=& [A_1, \ldots, A_n], \\
& & \quad \text{where } \begin{array}{ll} A_i = B_i, & \text{if } i \in |\mathcal{L}| \\ A_i = \rho_{C,i}, & \text{if } i \notin |\mathcal{L}| \end{array}
\end{array}
$$

$$
\begin{array}{rcl}
\rho^\dagger &=& [\rho^\star_{1,1}, \ldots, \rho^\star_{1,n}, \rho^\star_{0,1}, \ldots, \rho^\star_{0,n}] \\
|\{\ell_1, \ldots, \ell_n\}| &=& \{|\ell_1|, \ldots, |\ell_n|\}
\end{array}
$$

Figure 8. Type translation for simulating $\rightsquigarrow$ in $F_\omega^{\times+10}$

needed as each row variable may be used in records and variant types, which require different treatment for absent labels.

The next step is to handle list types. The rules involving comprehensions and unit lists (but not empty lists and concatenation) are the standard rules for monads. Benton et al [2] observe that the rules for monads can be straightforwardly simulated with those for sums by instantiating the monad constructor as the exception monad $T\,A = 1 + A$. We can do the same thing here (the monad constructor is called $List$ in our case). Even better, we can also give interpretations to the empty list and concatenation, such that the translation simulates the original rewrite rules.

The one missing ingredient is presence polymorphism. This is the one feature that makes use of type level computation. The idea is simple: encode presence types as type level booleans. We use the standard representation of booleans at the type level. Presence types have kind $\star \to \star \to \star$, where $\star$ is the kind of types. True ($\circ$) is represented as the type-level function $\lambda x^\star.\,\lambda y^\star.\,x$ and false ($\bullet$) as $\lambda x^\star.\,\lambda y^\star.\,y$. The conditional $\mathsf{if}\ P\ \mathsf{then}\ A\ \mathsf{else}\ B$ is represented as $P\,A\,B$.

The translation on types is given in Figure 8. It is parameterised by an environment σ that keeps track of which presence variables are present and which are absent. The translation on row types is additionally parameterised by a vector $\overline{B}$ that tracks the current row encoding, and a type C that is the current representation of an absent label (1 for a record and 0 for a variant).

The translation on terms is not difficult, but is more tedious. A minor technicality is that one needs to introduce $n-1$ dummy redexes for each projection and case. This is because the labels may need to be re-ordered. It is straightforward to verify that each $\rightsquigarrow$-reduction is simulated by one or more $F_\omega^{\times+10}$ reductions. □

In order to be sure that $norm$ is actually well-defined we need to ensure that $\rightsquigarrow$ is confluent.

Proposition 5. *The relation $\rightsquigarrow$ is confluent.*

Proof sketch. By an exhaustive case analysis we obtain weak confluence. Confluence follows from Theorem 4 and Newman's Lemma. □

The typing rule for query ensures that the input to the normalisation procedure has flat relation type and is tame, and can therefore be translated to an SQL query. Similarly, the typing rules for update and delete ensure that the where clauses and the set clause for update are tame and hence their normal forms can be directly translated to SQL expressions.

4.4 Query evaluation correctness

In this section we show that query evaluation is correct in the following sense: a program with query expressions (where queries are evaluated through normalising and then evaluating the query remotely on the database) is equivalent to an expression without query (where the equivalent processing is performed by loading the database tables into memory and evaluating the query code in-memory). The latter evaluation strategy is potentially much more inefficient, but provides a benchmark for the correctness of the former.

The first step is to show that $\rightsquigarrow$-reduction is sound with respect to the evaluation relation $\longrightarrow$. Given a database DB and a term M, let $inMem_{DB}(M)$ be M with all sub-terms of the form $\mathsf{table}\ t$ replaced by $asList(DB.t)$, all sub-terms of the form $\mathsf{asList}\ (N)$ replaced by N, and all table types $Table(S_r, S_w, S_n)$ replaced by $List\ \langle S_r \rangle$.

Lemma 6. *If $\vdash M : [\langle S \rangle]\,!\,\mathbf{wild}^\circ; E$ and $M \longrightarrow^* V$, then there exists $W \simeq V$ such that $inMem_{DB}(M) \longrightarrow^* W$.*

Proof. By induction on the structure of M. □

Lemma 7 (Soundness of $\rightsquigarrow$)**.** *If $\vdash M : [\langle S \rangle]\,!\,\mathbf{wild}^\circ; E$, $unQuery(M) \longrightarrow^* V$, and $unQuery(M) \rightsquigarrow N$, then there exists $W \simeq V$ such that $N \longrightarrow^* W$.*

Proof. By Lemma 6, we have $inMem_{DB}(unQuery(M)) \longrightarrow^* W$ with $W \simeq V$. As asList does not appear in the $\rightsquigarrow$-rules, we have $inMem_{DB}(unQuery(M)) \rightsquigarrow inMem_{DB}(N)$. Now all the applicable $\longrightarrow$-rules are also $\rightsquigarrow$-rules, so $inMem_{DB}(unQuery(M)) \rightsquigarrow^* W$. By confluence of $\rightsquigarrow$, we have $inMem_{DB}(N) \rightsquigarrow^* W$. Finally, as there are no free variables or asList constructors in $inMem_{DB}(N)$, we are free to choose a call-by-value reduction strategy that uses no commuting conversions, and hence $inMem_{DB}(N) \longrightarrow^* W$.

□

Lemma 8 (Soundness of splitting)**.**
If $\mathsf{for}\ (\overline{G})\ \mathsf{where}\ X\ M \longrightarrow^* V$, where $\vdash V : [\langle \overline{\ell : A} \rangle]\,!\,\mathbf{wild}^\circ; E$, and $(+\!+)\,\|M\|_{\overline{G},X} \longrightarrow^* W$, then $V \simeq W$.

Proof. By induction on the structure of M. □

Proposition 9. *If $\mathsf{query}\ M \longrightarrow^* V$, then there exists $W \simeq V$ such that $unQuery(M) \longrightarrow^* W$.*

Proof. By Lemmas 7 and 8. □

5. Links implementation

Up to now we have focused on CORELINKS as a core calculus for effects. We now consider some more practical issues by reference to the Links source language and implementation [17].

5.1 Source types

Effect type systems are widely studied, but rarely exposed in source languages. We believe that this may be largely because exposing effect types to the programmer can significantly complicate the types the programmer has to deal with. In particular, the standard extension of type inference with effect types includes a union operation on effects, allowing for an arbitrary number of effect variables to appear in any effect annotation. The use of effect rows amounts to introducing the restriction that only one effect variable can appear in each effect annotation.

The Links source language is based on a Hindley-Milner type system extended with rows and kinds. Type inference for polymorphic records and variants is standard. The extension to support effect rows is a straightforward application of the same technology.

5.2 Evaluation model

The *intermediate representation* (IR) used by Links is an A-normal form variant of CORELINKS. The server-side component of Links is implemented as a CPS interpreter for the IR. (CPS is used, amongst other things, for implementing web continuations and concurrency.) When the interpreter encounters a query expression it switches to a non-standard interpreter that implements the query normalisation rules as a big-step semantics and then sends the resulting query to the database.

Note that it is essential that queries be generated at run-time, because not enough information is available to generate them statically. Consider the *filterTable* example from the introduction. We cannot hope to generate a full query until the arguments are provided to the function, which we cannot expect to happen in general until run-time, as the arguments can depend on arbitrary computation. In this simple case we might hope to normalise most of the component parts in advance, which might be seen as an optimisation. However, suppose we were to abstract further over the data source, or build up more complex queries by performing joins and abstracting over other data sources, then a non-trivial amount of normalisation would inevitably remain to do at run-time.

5.3 Extensions

Links supports some extensions to CORELINKS. It is possible to attach an orderby clause to a for comprehension. This corresponds to the SQL order by clause. For ordering to be meaningful it is necessary to move from a bag to a list semantics. However, though our rewrite relation $\rightsquigarrow$ is sound for lists, the splitting function is not. The development version of Links implements a more sophisticated splitting function that accommodates a list semantics. It inserts special ordering indexes in order to determinise the output.

To allow sorting on fields from multiple tables Links generalises comprehensions to support multiple generators. In order to effectively desugar such generalised for comprehensions we use a variant of the 'comprehensive comprehensions' compilation scheme introduced by Peyton Jones and Wadler [16]. This scheme allows us to desugar for comprehensions using `map`, `concatMap` and `sortBy` functions (being careful to ensure that these functions are specially annotated as tame). The non-standard interpreter then rediscovers the generalised for comprehension structure as part of its big-step normalisation procedure. This turns out to be surprisingly straightforward. This design has the pleasing property that it does not require us to complicate the IR by the inclusion of for comprehensions (which makes targetting environments other than the database considerably simpler than it might be).

5.4 Recursive types

Links supports recursive types. Unfortunately this exposes a hole in the effect-type system, as it is possible to encode the Y-combinator and hence write non-terminating queries. We could certainly plug this hole by partitioning recursive types into positive and negative ones, and associating the **wild** effect with the elimination form for negative recursive types. We have not yet implemented such a system, partly because we are concerned about the negative impact it may have on the usability of the rest of the language. In practice, we have not had a problem with negative recursive types leading to non-termination. It seems considerably less likely for a programmer to accidentally encode a non-terminating term using negative recursive types than to accidentally write a call to a standard recursive function inside a query. Nevertheless, this area deserves further investigation.

5.5 Syntactic sugar for functions

As language designers one of our goals has been to try to support effect types in a way that is not too intrusive to the programmer. A glance at the typing rules might suggest that CORELINKS types are too verbose. For instance, every recursive function has a **wild** annotation. We have ameliorated this issue somewhat by introducing special syntactic sugar for the two effects we currently support.

The CORELINKS type $A \to^E B$ is written in Links as `(A) {E}-> B` (Links functions are n-ary; hence argument type(s) are written in parentheses). The $\cdot$ at the end of a closed row is never written explicitly. Row type extension $\ell; R$ is written `l | R`. An empty open row ρ is written `|rho`. The syntactic sugar for function arrows is summarised as follows:

```
-e->          ≡ { |e}->
->            ≡ -e->, e fresh
{E}~>         ≡ {wild | E}->
~e~>          ≡ { |e}~>
~>            ≡ ~e~>, e fresh
{:A | E}~>    ≡ {hear:A | E}~>
```

The `->` suffix indicates a tame arrow and the `~>` suffix indicates a wild arrow. Type variables are written in the middle of an arrow. The `hear:A` effect is abbreviated `:A`. First order functions that do not receive messages always have arrows of the form `->` or `~>`. We can always assign fresh effect variables to the first n arguments of a curried function (as partially applying a curried function has no effects). Interestingly, many higher-order functions (at least, most of the standard ones included in the Links prelude) follow a consistent pattern: function arguments are all assigned the same effect as the function body. This suggests that it might be worth giving `->` and `~>` more sophisticated meanings in order to make higher-order function types more readable. For example, the type inferred for the `uncurry` function is:

```
((a) -b-> (c) -b-> d) -> (a, c) -b-> d
```

Because this follows the common pattern, we might hope for this to be displayed instead as:

```
((a) -> (b) -> c) -> (a, b) -> c
```

Effect rows scale to support unbounded numbers of effects. For instance, Blume et al. [4] use them to track an arbitrary number of exceptions. However, it is not clear how acceptable the resulting verbosity is to programmers. One could imagine introducing some kind of support from an IDE for selectively hiding some of the effect annotations on types.

6. Related work

Our work builds on a long line of research on functional database query languages, comprehensions, and language-integrated querying. Links was explicitly inspired by work on Kleisli [27], a functional query language; moreover, Cooper's work on higher-order query normalisation [10] (which we refine in this paper) builds on Wong's seminal work on conservativity and query normalisation [26]. Subsequently, Links has co-evolved with the LINQ project at Microsoft, which has brought the benefits of language-integrated queries and comprehension syntax to a wide audience.

Row type systems have been widely studied [6, 15, 20, 22, 25]. The basis for our row type system is Remy's $\mathbf{\Pi ML}'$ [22]. Our typing rules for polymorphic variants are similar to those of Blume et al. [3]. Unlike their system CORELINKS does not support first-class cases, but the full Links does allow them to be implemented as functions using upcasts.

The Ur/Web project [8] uses advanced record typing features, along with powerful generative programming techniques, to type-check SQL primitives embedded in a high-level functional language. This approach provides greater control over what query code is generated, but makes an explicit distinction between query code and ordinary code.

SML# [21] extends Standard ML with polymorphic records and support for SQL. Like Ur/Web it makes an explicit distinction between query code and ordinary code.

Ferry [13] is a functional query language that provides nested data, grouping, and aggregation, and supports list semantics rather than SQL's multiset semantics. It translates to SQL via a monolithic translation to a flat relational algebra which comes equipped with a sophisticated optimisation engine. Like Links query expressions, Ferry programs that return a list of records of base type are guaranteed to be translated to at most one SQL query. Ferry programs can also return nested data. In this case an additional collection of SQL queries, whose size is equal to the number of nested lists in the type of the program, is generated. Ulrich [23] has developed a variant of Links that builds on Ferry. It integrates smoothly with the Links type system, but uses Ferry as a back-end instead of our query normaliser. A difficulty with the Ferry approach is that it is rather hard to reason about due to the monolithic translation.

7. Conclusions and future work

Our experience with Links [7] has shown that our effect system does appear to work well in practice, and it is a significant advantage to use standard row and polymorphic type inference mechanisms for effects. Nevertheless there may still be room for improvement in the design of CORELINKS. The particular form of presence polymorphism we support is rather unusual, and can be counterintuitive (if a label is absent, then it seems strange to have to give it a type). Alternatives include using explicit constraints alongside types along the lines of OCaml or SML#, or some combination of rows and constraints, or even moving to a more expressive type system supporting refinement types or more general dependent types.

Much work remains in extending support for database functionality. The fragment of SQL currently targeted is limited. For instance, it does not support null values, grouping or aggregation, all of which are key parts of SQL. We believe our approach is complementary to ongoing work in Ur/Web and Ferry: it would be interesting to see whether Links' implicit query normalisation approach can be combined with the explicit approach in Ur/Web. We are currently working on establishing type soundness for concurrency in Links using row-based effects and on developing a rewriting-based technique for evaluating Ferry-style nested queries using multiple flat queries.

An important concern when adding new effects to a source language is how this will affect the usability and readability of the language. It would be interesting to see how well our approach scales to support more effects, and to what extent it is possible to balance the concerns of providing precise effect information in types while keeping types readable.

Acknowledgments

We would like to thank Dan Licata and the reviewers for their constructive feedback, Andreas Abel for pointing us at Ralph Matthes's strong normalisation results for extensions of System F [18], and Nicolas Oury for helpful discussions about encoding row types. This work was supported by a Google Research Award.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice Hall International, 1996.

[2] P. N. Benton, G. M. Bierman, and V. de Paiva. Computational types from a logical perspective. *J. Funct. Program.*, 8(2):177–193, 1998.

[3] M. Blume, U. A. Acar, and W. Chae. Extensible programming with first-class cases. In *ICFP*, pages 239–250, 2006.

[4] M. Blume, U. A. Acar, and W. Chae. Exception handlers as extensible cases. In *APLAS*, pages 273–289, 2008.

[5] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[6] L. Cardelli and J. C. Mitchell. Operations on records. In Gunter and Mitchell [14], pages 295–350.

[7] J. Cheney, S. Lindley, and H. Müller. DBWiki: A database wiki prototyped in Links. In *DBPL*, 2011.

[8] A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *PLDI*, pages 122–133, 2010.

[9] E. Cooper. *Programming language features for web application development*. PhD thesis, University of Edinburgh, 2009.

[10] 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.

[11] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: web programming without tiers. In *FMCO*, volume 4709 of *LNCS*, pages 266–296, 2007.

[12] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, Sept. 1998.

[13] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: database-supported program execution. In *SIGMOD Conference*, pages 1063–1066, 2009.

[14] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. Foundations of Computing Series. MIT Press, 1993.

[15] R. Harper and B. C. Pierce. A record calculus based on symmetric concatenation. In *POPL*, pages 131–142, 1991.

[16] S. P. Jones and P. Wadler. Comprehensive comprehensions. In *Haskell Workshop*, pages 61–72, 2007.

[17] Links. `http://groups.inf.ed.ac.uk/links`.

[18] R. Matthes. Non-strictly positive fixed points for classical natural deduction. *Ann. Pure Appl. Logic*, 133(1-3):205–230, 2005.

[19] H. R. Nielson, F. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In *LOMAPS*, 1996.

[20] A. Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895, 1995.

[21] A. Ohori and K. Ueno. Making Standard ML a practical database programming language. In *ICFP*, pages 307–319, 2011.

[22] D. Rémy. Type inference for records in a natural extension of ML. In Gunter and Mitchell [14], pages 67–95.

[23] A. Ulrich. A Ferry-based query backend for the Links programming language. Master's thesis, University of Tübingen, 2011.

[24] P. Wadler and P. Thiemann. The marriage of effects and monads. *Transactions on Computational Logic*, 4(1):1–32, 2003.

[25] M. Wand. Complete type inference for simple objects. In *LICS*, pages 37–44, 1987.

[26] L. Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3):495–505, 1996.

[27] L. Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1):19–56, 2000.