Multi-Model Query Compilation Using Bi-Algebra Comprehensions

Ryan Wisnesky
Conexus AI

September 2, 2020

Abstract

In this paper we extend the monad comprehension calculus, a well-studied user-facing query formalism sufficient to encode (nested) relational algebra and admitting an easily-optimized iterator-based execution model, to arbitrary user-defined abstract data types. Following Erwig, our formalism is parameterized by a bi-cartesian category L representing an ambient type theory and we represent an implementation of abstract data type F as an F-bi-algebra on L. Using the internal language of the free bi-cartesian category, we define L-independent implementations of bi-algebras for sets, graphs, infinite streams, finite maps, state, and people (representing a business object), and demonstrate how to translate bi-algebras from one choice of L to another; translation from one bi-algebra to another within the same L is accomplished with natural transformations in the usual way.

1 Introduction

In this paper we are concerned with three recent trends in database systems and theory:

- the development, beginning in the 1990s [19, 11], of query formalisms, compilers, and runtimes based on the interplay between structural recursion and comprehension syntax, as exemplified by systems such as Microsoft's Dyrad-LINQ [20] and Data Parallel Haskell [5]; and
- the development, beginning in the 2000s, of data integration formalisms based on first-order logic, used to mediate between semantically heterogeneous database schemas, as exemplified in systems such as IBM's Clio [12] and Microsoft's Rondo [13]; and
- the development, beginning in the 2010s, of "big data" query formalisms, compilers, and runtimes based on graph walks over various distributed data stores, as exemplified by systems such as Apache Tinkerpop [14] and foreshadowed by mixed SQL/Hadoop runtimes such as Hadapt [3].

The above trends are related to each other as follows:

- although query compilers based on structural recursion and comprehension syntax are often used in relational data integration systems, including Clio and Rondo, using them with graph databases runs into non-trivial problems [8] with purely functional graph representations; and
- although data integration formalisms based on first-order logic can mediate between schemas in the same data model, using them to mediate between data models themselves runs into non-trivial problems with representing logic using logic [1]; and
- although graph-based "big data" systems such Apache TinkerPop organically developed techniques similar to structural recursion, comprehension syntax and logic-based schema mappings, because of the problems described above, many of these techniques are purely informal, or formalized operationally [14].

1.1 Contributions

In this paper we address the challenges above by extending the monad comprehension calculus, a well-studied user-facing query formalism sufficient to encode (nested) relational algebra and admitting an easily-optimized iterator-based execution model, to arbitrary user-defined abstract data types. Following Erwig [7], our formalism is parameterized by a bi-cartesian category L, representing an ambient type theory, and we represent an implementation of abstract data type F as an F-bi-algebra on L. After reviewing the internal language of bi-cartesian categories, we use that syntax to define L-independent implementations of bi-algebras for sets, graphs, infinite streams, finite maps, state, and people (representing a business object), and demonstrate how to translate bi-algebras from one choice of L to another; translating from one bi-algebra to another within the same L is accomplished with natural transformations in the usual way. Our formalism has many uses:

- to generalize compilers that are based on structural recursion and comprehensions, such as LINQ [20] and Data Parallel Haskell [5], to handle graphs; and
- to generalize data integration formalisms based on first-order logic to account for morphisms between underlying data models, such as between graphs and relations; and
- to provide a rigorous mathematical foundation / intermediate form for existing graph processing systems, such as Apache Tinkerpop [14].

To exemplify these uses, we prove theorems relating our formalism to that of algebraic databases [16] and algebraic property graphs [17], demonstrate how our formalism can interpret the Apache Tinkerpop Gremlin family of languages [14], and conclude with a discussion of current work on distributed runtimes for our formalism .

1.2 Outline

This paper is structured as follows:

- In section 2, we describe a standard formalism (which we call CAML) for query processing with structural recursion and co-recursion, similar to Grust's monad comprehension calculus [11, 19, 18].
- In section 3, we describe how to extend CAML to allow for arbitrary user-defined abstract data types, following Erwig's bi-algebra approach [9, 7], and we describe how to migrate bi-algebras from one programming language to another by way of CAML.
- In section 4, we implement a set of bi-algebras and morphisms for common abstract data types such as sets and graphs, following Erwig's inductive graph approach [8], and demonstrate how to interpret a fragment of Apache Tinkerpop Gremlin [14] in our formalism.
- In section 5, we relate our formalism to those of algebraic databases [16] and property graphs [17].

We conclude in section 6 with some discussion on distributed runtimes for our formalism. Knowledge of type theory and category theory at the level of [2] and knowledge of functional query languages at the level of [11] are required to understand this paper.

2 Categorical Abstract Machine Language (CAML)

In this section we describe a minimal, standard formalism (which we call CAML) for query processing with structural recursion on finite lists and structural co-recursion on infinite streams, similar to Wong's nested relational calculus [19], Grust's monad comprehension calculus [11], Curien's Categorical Abstract Machine Language [6], and many others.

2.1 Syntax

A *type* t is defined as a term in the context-free grammar:

$$t ::= \mathsf{Int} \mid 0 \mid 1 \mid t + t \mid t \times t \mid t \Rightarrow t \mid \mathsf{List}\ t \mid \mathsf{Stream}\ t$$

A $(t, t'$-typed) *expression* $e : t \to t'$ is defined as a term in the context-free grammar (where we omit the | symbols between productions on different lines):

$$e : t \to t' \quad ::= \quad \mathsf{id}_t : t \to t \;\mid\; (e' : t' \to t'') \circ (e : t \to t') : t \to t''$$
$$\mathsf{const}_c\ (c \in \{0, 1, -1, \ldots\}) : 1 \to \mathsf{Int} \;\mid\; \mathsf{add}, \mathsf{mul} : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int} \;\mid\; \mathsf{neg} : \mathsf{Int} \to \mathsf{Int}$$
$$\mathsf{eq}_t : t \times t \to 1 + 1 \;\mid\; \delta_{t,t',t''} : t \times (t' + t'') \to (t \times t') + (t \times t'')$$
$$\mathsf{tt}_t : t \to 1 \;\mid\; \mathsf{fst}_{t,t'} : t \times t' \to t \;\mid\; \mathsf{snd}_{t,t'} : t \times t' \to t' \;\mid\; \langle e : t \to t' \ , \ e' : t \to t'' \rangle : t \to t' \times t'$$
$$\mathsf{ff}_t : 0 \to t \;\mid\; \mathsf{inl}_{t,t'} : t \to t + t' \;\mid\; \mathsf{inr}_{t,t'} : t' \to t + t' \;\mid\; \langle e : t' \to t \mid e' : t'' \to t \rangle : t' + t'' \to t$$
$$\Lambda(e : t \times t' \to t'') : t \to t' \Rightarrow t'' \;\mid\; \mathsf{eval}_{t,t'} : (t' \Rightarrow t) \times t' \to t$$
$$\mathsf{in}_t : 1 + (t \times \mathsf{List}\ t) \ \to \ \mathsf{List}\ t \quad \mid \quad \mathsf{fold}(e : \tau \times (1 + (t \times t')) \to t') : \tau \times \mathsf{List}\ t \ \to \ t'$$
$$\mathsf{out}_t : \mathsf{Stream}\ t \ \to \ t \times \mathsf{Stream}\ t \;\mid\; \mathsf{unfold}(e : t \to t' \times t) : t \to \mathsf{Stream}\ t'$$

We abbreviate:

$$\mathsf{Bool} := 1+1 \quad \mathsf{true}_t := \mathsf{inl} \circ \mathsf{tt}_t \quad \mathsf{false}_t := \mathsf{inr} \circ \mathsf{tt}_t \quad f \otimes g := \langle f \circ \mathsf{fst} \ , \ g \circ \mathsf{snd} \rangle \quad f \oplus g := \langle \mathsf{inl} \circ f \mid \mathsf{inr} \circ g \rangle$$
$$\mathsf{fold}'(f) := \mathsf{fold}(f \circ \mathsf{snd}) \circ \langle \mathsf{tt}, \mathsf{id} \rangle \quad \mathsf{unfold}'(f) := \mathsf{unfold} \langle \mathsf{fst} \circ f, \langle \mathsf{fst}, \mathsf{snd} \circ f \rangle \rangle$$

As an example, to sum the length of a list we may write: $\mathsf{fold}' \langle \mathsf{const}_0 \mid \mathsf{add} \rangle : \mathsf{List}\ \mathsf{Int} \to \mathsf{Int}$.

So that our formalism can be used without function types, we use parameterized folds, which additionally require an input parameter of type τ compared to the form usually seen in the literature; parameterized folds can be defined in terms of the other expressions, but only in a higher-order setting (requiring eval and Λ). Similarly, the distributivity expression δ can be defined in terms of the other expressions, but only in a higher-order setting, because not all bi-cartesian categories are distributive, but all closed ones are [10].

The fragment of CAML that does not contain function types or stream types can express exactly the primitive recursive functions on lists; dually, the fragment of CAML that does not contain function types or list types can express exactly the primitive co-recursive function on streams; in the presence of function types, CAML can express non-primitive-recursive functions on lists and streams, such as the Ackermann's function.

2.2 Axiomatic Semantics

The axiomatic semantics of CAML is defined as the smallest equivalence relation $\approx$ that is also a congruence (i.e., for which $f \approx f'$ implies $g \circ f \approx g \circ f'$, etc):

$$\mathsf{id} \circ f \approx f \quad f \circ \mathsf{id} \approx f \quad f \circ (g \circ h) \approx (f \circ g) \circ h \quad \mathsf{eq} \circ \langle e, f \rangle \approx \mathsf{true} \text{ iff } e \approx f$$
$$f : t \to 1 \approx \mathsf{tt}_t \quad \mathsf{fst} \circ \langle f, g \rangle \approx f \quad \mathsf{snd} \circ \langle f, g \rangle \approx g \quad \langle \mathsf{fst} \circ f, \ \mathsf{snd} \circ f \rangle \approx f$$
$$f : 0 \to t \approx \mathsf{ff}_t \quad \langle f | g \rangle \circ \mathsf{inl} \approx f \quad \langle f | g \rangle \circ \mathsf{inr} \approx g \quad \langle f \circ \mathsf{inl} \mid f \circ \mathsf{inr} \rangle \approx f$$
$$\mathsf{eval} \circ (\Lambda(f) \otimes \mathsf{id}) \approx f \quad \Lambda(\mathsf{eval} \circ (g \otimes \mathsf{id})) \approx g \quad \langle \mathsf{inl} \circ \mathsf{snd} \mid \mathsf{inr} \circ \mathsf{snd} \rangle \circ \delta \approx \mathsf{snd} \quad \langle \mathsf{fst} \mid \mathsf{fst} \rangle \circ \delta \approx \mathsf{fst}$$
$$\delta \circ (\mathsf{id} \otimes \mathsf{inl}) \approx \mathsf{inl} \quad \delta \circ (\mathsf{id} \otimes \mathsf{inr}) \approx \mathsf{inr} \quad \delta \circ (f \otimes \langle \mathsf{inr} \circ g \mid \mathsf{inl} \ \circ h \rangle) \approx \langle \mathsf{inr} \circ (f \otimes g) \mid \mathsf{inl} \circ (f \otimes h) \rangle \circ \delta$$
$$\mathsf{fold}(f) \circ (\mathsf{id} \otimes \mathsf{in}_t) \approx f \circ \langle \mathsf{fst}, (\mathsf{tt}_1 \oplus (\mathsf{id}_t \otimes \mathsf{fold}(f))) \circ \alpha \rangle \quad g \approx \mathsf{fold}(f) \text{ if } g \circ (\mathsf{id} \otimes \mathsf{in}_t) \approx f \circ \langle \mathsf{fst}, (\mathsf{tt}_1 \oplus (\mathsf{id}_t \otimes g)) \circ \alpha \rangle$$
$$g \approx \mathsf{unfold}(f) \text{ if } g \circ \mathsf{unfold}(f) \approx ((\mathsf{id}_t \otimes g)) \circ f \quad \mathsf{out}_t \circ \mathsf{unfold}(f) \approx (\mathsf{id}_t \otimes \mathsf{unfold}(f)) \circ f$$

with isomorphism: $\alpha : \Gamma \times (1 + (A \times B)) \to 1 + (A \times (\Gamma \times B))$ and equations stating that Int forms a commutative ring in the obvious way, our choice of base types, primitives, and their axioms being ad-hoc in this paper.

The *syntactic category* CAML is defined as the category formed by taking types as objects and equivalence classes of expressions modulo $\approx$ as morphisms. These axioms, minus lists, streams, arithmetic, and equality, were shown to be decidable in [15].

Remark on Type Isomorphism. Product and sum types have the pleasing property of admitting a complete (and decidable) axiomatization as the least congruence $\sim$ for which $(0, +)$ and $(1, \times)$ are commutative monoids, $\times$ distributes over $+$, and $t \times 0 \sim 0$. This axiomatization allows mediating between isomorphic product and sum types automatically, a common operation in query processing. Surprisingly, although the type constructor $\Rightarrow$ satisfies Tarski's "high school algebra axioms" such as $a \Rightarrow b \times c \sim (a \Rightarrow b) \times (a \Rightarrow c)$, type isomorphism with $+$ and $\times$ and $\Rightarrow$ together is not recursively axiomatizable, and its decidability is unknown [10].

2.3 Denotational Semantics

The set-theoretic semantics $[\![t]\!]$ types t is standard and can be recursively defined as a functor from CAML to the category of sets:

- Int denotes the set of integers, $\{0, 1, -1, 2, -2, \ldots\}$.
- $\mathsf{0}$ denotes the empty set $\{\}$, also written $\emptyset$, and $\mathsf{1}$ denotes the singleton set $\{\emptyset\}$.
- $t \times t'$ denotes the Cartesian product of the sets $[\![t]\!]$ and $[\![t']\!]$, where e.g,

$$\{1, 2\} \times \{2, 3\} = \{(1, 2), (1, 3), (2, 2), (2, 3)\}.$$

- $t + t'$ denotes the disjoint union of the sets $[\![t]\!]$ and $[\![g']\!]$, where e.g,

$$\{1, 2\} + \{2, 3\} = \{(\emptyset, 1), (\emptyset, 2), (\{\emptyset\}, 2), (\{\emptyset\}, 3)\},$$

  where we have chosen $\emptyset$ and $\{\emptyset\}$ as left and right "labels".
- $t \Rightarrow t'$ denotes the set of functions from $[\![t]\!]$ to $[\![t']\!]$.
- $\mathsf{List}\ t$ denotes the set of finite lists of $[\![t]\!]$.
- $\mathsf{Stream}\ t$ denotes the set of infinite lists (so-called *streams*) with elements in $[\![t]\!]$.

The set-theoretic semantics $[\![e]\!]$ of an expression $e : t \to t'$ is a function $[\![t]\!] \to [\![t']\!]$ is also standard:

- $\mathsf{id}_t : t \to t$ denotes the identity function on $[\![t]\!]$, and $f \circ g$ denotes the function composition of $[\![f]\!]$ with $[\![g]\!]$; i.e., $[\![f \circ g]\!](x) := [\![f]\!]([\![g]\!](x))$.
- $\mathsf{const}_c : \mathsf{1} \to \mathsf{Int}$ denotes the constant function returning integer c, and $\mathsf{add}, \mathsf{mul} : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$ denote integer addition and multiplication, respectively, and $\mathsf{neg} : \mathsf{Int} \to \mathsf{Int}$ denotes negation (subtraction from zero).
- $\mathsf{eq}_t : t \times t \to \mathsf{Bool}$ denotes the characteristic function of $\{(x, x) \mid x \in [\![t]\!]\}$; i.e., the $[\![t]\!]$-equality function.
- $\mathsf{tt}_t : t \to \mathsf{1}$ denotes the function sending any element of $[\![t]\!]$ to $\emptyset$, and $\mathsf{ff}_t : \mathsf{0} \to t$ denotes the only function from $\emptyset$ to $[\![t]\!]$.
- $\mathsf{fst}_{t,t'}$ and $\mathsf{snd}_{t,t'}$ denote the first and second projections from $[\![t \times t']\!]$, respectively, and $\langle f : t \to t', g : t \to t'' \rangle$ denotes the function taking each $x \in [\![t]\!]$ to the pair $([\![f]\!](x), [\![g]\!](x))$.
- $\mathsf{inl}_{t,t'}$ and $\mathsf{inr}_{t,t'}$ denote the first and second injections to $[\![t + t']\!]$, respectively, and $\langle f : t' \to t \mid g : t'' \to t \rangle$ denotes the case-analysis function taking each $(\emptyset, x) \in [\![t' + t'']\!]$ to $[\![f]\!](x)$ and and each $(\{\emptyset\}, y) \in [\![t' + t'']\!]$ to $[\![g]\!](y)$.
- $\delta_{t,t',t''} : t \times (t' + t'') \to (t \times t') + (t \times t'')$ denotes the function taking $(x, (l, y))$ to $(l, (x, y))$.
- $\Lambda(e : t \times t' \to t'') : t \to t' \Rightarrow t''$ denotes the "currying" function taking each $x \in [\![t]\!]$ to the function taking each $y \in [\![t']\!]$ to $[\![e]\!](x, y)$, and $\mathsf{eval}_{t,t'}$ denotes the function taking $(f : [\![t']\!] \to [\![t]\!], g : [\![t']\!])$ to $f(g)$.
- in_t denotes the initial algebra for lists of type t in insert presentation, and $\mathsf{fold}(e : \tau \times (\mathsf{1} + t \times t') \to t') : \tau \times \mathsf{List}\ t \to t'$ denotes structural iteration with a parameter of type τ.
- $\mathsf{out}_{\tau,t}$ denotes the final co-algebra for streams in insert presentation, and $\mathsf{unfold}(e : t' \to t \times t') : t' \to \mathsf{Stream}\ t$ denotes structural co-iteration .

Theorem 1. If $f, g : t \to t'$ are such that $f \approx g$, then $[\![f]\!] = [\![g]\!]$ as set-theoretic functions $[\![t]\!] \to [\![t']\!]$.
Proof. In Coq, by induction on $f \cong g$. □

2.4 Lambda calculus Form

A more user-friendly syntax for CAML than the variable-free form above is that of a λ-calculus [6] that allows typed variables, which we write as $v{:}t$ for variable v and type t. A *context* is a list of distinct variable-type pairs:

$$\Gamma \ ::= \ - \mid \Gamma, v{:}t \ (v \notin \Gamma)$$

A *typing judgment*, also called an *expression-in-context*, $\Gamma \vdash e : t$ is a relation on contexts Γ, terms e, and types t and is inductively defined using the inference rules:

$$\frac{\Gamma \vdash v : t \qquad v \neq v'}{\Gamma, v'{:}t' \vdash v : t} \qquad \frac{}{\Gamma, v{:}t \vdash v : t} \qquad \frac{c \in \{..., 0, ...\}}{\Gamma \vdash \mathtt{K}_c : \mathsf{Int}} \qquad \frac{\Gamma \vdash e : t \qquad \Gamma \vdash f : t}{\Gamma \vdash \mathtt{eq}\ e\ f : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathtt{tt} : 1}$$

$$\frac{\Gamma \vdash e : t \times t'}{\Gamma \vdash \mathtt{fst}\ e : t} \qquad \frac{\Gamma \vdash e : t \times t'}{\Gamma \vdash \mathtt{snd}\ e : t'} \qquad \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e' : t'}{\Gamma \vdash (\!| e, e' |\!) : t \times t'} \qquad \frac{\Gamma \vdash e : \mathsf{Int} \qquad \Gamma \vdash e' : \mathsf{Int}}{\Gamma \vdash e +/* e' : \mathsf{Int}}$$

$$\frac{\Gamma \vdash e : \mathsf{Int}}{\Gamma \vdash - e : \mathsf{Int}} \qquad \frac{\Gamma \vdash e : 0}{\Gamma \vdash \mathtt{ff}_t\ e : t} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathtt{inl}_t\ e : t' + t} \qquad \frac{\Gamma \vdash e : t'}{\Gamma \vdash \mathtt{inr}_t\ e : t + t'}$$

$$\frac{\Gamma, v : t \vdash e'' : t'' \qquad \Gamma, v : t' \vdash e' : t'' \qquad \Gamma \vdash e : t + t' \qquad v \notin \Gamma}{\Gamma \vdash \mathtt{case}_v\ e\ \mathtt{of}\ e' \mid e''\ : t''} \qquad \frac{\Gamma, v{:}t \vdash e : t' \qquad v \notin \Gamma}{\Gamma \vdash \lambda v{:}t.e : t \Rightarrow t'}$$

$$\frac{\Gamma \vdash e' : t' \qquad \Gamma \vdash e : t' \Rightarrow t}{\Gamma \vdash e e' : t} \qquad \frac{\Gamma \vdash e : 1 + t \times \mathsf{List}\ t}{\Gamma \vdash \mathtt{in}\ e : \mathsf{List}\ t} \qquad \frac{\Gamma \vdash e : \mathsf{Stream}\ t}{\Gamma \vdash \mathtt{out}\ e : t \times \mathsf{Stream}\ t}$$

$$\frac{\Gamma, v : 1 + t \times t' \vdash e : t' \qquad v \notin \Gamma \qquad \Gamma \vdash f : \mathsf{List}\ t}{\Gamma \vdash \mathtt{fold}\ (\lambda v.e)\ f : t'} \qquad \frac{\Gamma, v : t \vdash e : t \times t' \qquad v \notin \Gamma \qquad \Gamma \vdash f : t}{\Gamma \vdash \mathtt{unfold}\ (\lambda v.e)\ f : \mathsf{Stream}\ t'}$$

We have altered the typeface of keywords above to ensure that they are distinct from the combinator form of CAML. The translation from the above syntax to the variable-free syntax is standard: contexts are translated into product types:

$$[-] := 1 \quad [\Gamma, v : t] := [\Gamma] \times t$$

and a typing judgment $\Gamma \vdash e : t$ is translated into an expression $[e] : [\Gamma] \to [t]$ in combinator form by translating each variable into a projection:

$$[\Gamma, v' : t' \vdash v : t] := [\Gamma \vdash v : t] \circ \mathsf{fst}_{[\Gamma],[t]} \quad [\Gamma, v : t \vdash v : t] := \mathsf{snd}_{[\Gamma],[t]} \quad [\Gamma \vdash \mathtt{K}_c : \mathsf{Int}] := \mathsf{const}_c \circ \mathsf{tt}_{[\Gamma]}$$

$$[\Gamma \vdash \mathtt{eq}\ e\ f : \mathsf{Bool}] := \mathsf{eq} \circ \langle [e], [f] \rangle \quad [\Gamma \vdash e +/* e' : \mathsf{Int}] := \mathsf{plus/mul} \circ \langle [e], [f] \rangle \quad [\Gamma \vdash -e : \mathsf{Int}] := \mathsf{neg} \circ [e]$$

$$[\Gamma \vdash \mathtt{fst}\ e : t] := \mathsf{fst}_{[t],[t']} \circ [\Gamma \vdash e : t \times t'] \quad [\Gamma \vdash \mathtt{snd}\ e : t'] := \mathsf{snd}_{[t],[t']} \circ [\Gamma \vdash e : t \times t']$$

$$[\Gamma \vdash (\!| e, e' |\!) : t \times t'] := \langle [\Gamma \vdash e : t], [\Gamma \vdash e' : t'] \rangle \quad [\Gamma \vdash \mathtt{tt} : 1] := \mathsf{tt}_{[\Gamma]} \quad [\Gamma \vdash \mathtt{ff}\ e : t] := \mathsf{ff}_{[t]} \circ [\Gamma \vdash e : 0]$$

$$[\Gamma \vdash \mathtt{case}_v\ e\ \mathtt{of}\ e' \mid e'' : t''] := \langle [\Gamma, v{:}t \vdash e'{:}t''] | [\Gamma, v{:}t' \vdash e''{:}t''] \rangle \circ \delta \circ \langle \mathsf{id}_{[\Gamma]}, [\Gamma \vdash e{:}t + t'] \rangle$$

$$[\Gamma \vdash \mathtt{inl}_{t'}\ e : t + t'] := \mathsf{inl}_{[t],[t']} \circ [\Gamma \vdash e : t] \quad [\Gamma \vdash \mathtt{inr}_{t'}\ e : t'] := \mathsf{inr}_{[t],[t']} \circ [\Gamma \vdash e : t']$$

$$[\Gamma \vdash \lambda x : t.e : t \Rightarrow t'] := \Lambda[\Gamma, x : t \vdash e : t'] \quad [\Gamma \vdash e' e : t'] := \mathsf{eval} \circ \langle [\Gamma \vdash e' : t \Rightarrow t'], [\Gamma \vdash e : t] \rangle$$

$$[\Gamma \vdash \mathtt{in}\ e : \mathsf{List}\ t] := \mathsf{in}_{[t]} \circ [\Gamma \vdash e : 1 + (t \times \mathsf{List}\ t)] \quad [\Gamma \vdash \mathtt{out}\ e : t \times \mathsf{List}\ t] := \mathsf{out}_{[t]} \circ [\Gamma \vdash e : \mathsf{Stream}\ t]$$

$$[\Gamma \vdash \mathtt{fold}\ (\lambda v.e)\ f : t'] := \mathsf{fold}_{\Gamma,t,t'}\ [\Gamma, v : 1 + (t \times t') \vdash e : t'] \circ \langle \mathsf{id}_{[\Gamma]}, [\Gamma \vdash f : \mathsf{List}\ t] \rangle$$

$$[\Gamma \vdash \mathtt{unfold}\ (\lambda v.e)\ f : \mathsf{Stream}\ t] := \mathsf{unfold}'_{\Gamma,t,t'}\ [\Gamma, v : t \times t' \vdash e : t \times t'] \circ \langle \mathsf{id}_{[\Gamma]}, [\Gamma \vdash f : \mathsf{Stream}\ t] \rangle$$

2.5 The Set Monad

Structural recursion (and co-recursion) has much to recommend it as in intermediate form for query compilation. In particular, folds (and unfolds) are easily optimized [11]. However, we run into complications when generalizing fold to *non-free* collection types. For example, although we may represent sets as "lists up to permutation", to do so we require that whenever we fold a function f through a list l representing a set, for every permutation l' of l, that $f(l) = f(l')$, a potentially undecidable property. For this reason, a restricted form of fold, known as *comprehension*, is often proposed as a query language for non-free inductive datatypes, which then compiles into structural recursion in a way known to respect the defining equations of the non-free type, such as invariance under permutation.

To exemplify such non-free types, we add types and expressions for the *set monad* to CAML. The types and expressions of CAML are extended with:

$$t \quad ::= \quad \ldots \quad | \quad \mathsf{Set}\ t$$

$$e \quad ::= \quad \ldots \ | \ \eta_t : t \to \mathsf{Set}\ t \ | \ \varnothing_t : 1 \to \mathsf{Set}\ t \ | \ \Upsilon_t : \mathsf{Set}\ t \times \mathsf{Set}\ t \to \mathsf{Set}\ t \ | \ \mu_t : \mathsf{Set}\ (\mathsf{Set}\ t) \to \mathsf{Set}\ t$$
$$\mathsf{map}(e : t \to t') : \mathsf{Set}\ t \to \mathsf{Set}\ t' \ | \ \rho_{t,t'} : t \times \mathsf{Set}\ t' \to \mathsf{Set}\ (t \times t')$$

and the axiomatic semantics with:

$$\mathsf{map}(\mathsf{snd}) \circ \rho \approx \mathsf{snd} \quad \rho \circ (\mathsf{id} \otimes \eta) \approx \eta \mathsf{map}(f) \circ \varnothing' \approx \varnothing' \quad \rho \circ \langle e, \varnothing' \rangle \approx \varnothing' \quad \mu \circ \varnothing' \approx \varnothing'$$
$$\rho \circ (\mathsf{id} \otimes \mu) \approx \mu \circ \mathsf{map}(\rho) \circ \rho \quad \mathsf{map}(f \otimes g) \circ \rho \approx \rho \circ (f \otimes \mathsf{map}(g)) \quad \mathsf{map}(\theta) \circ \rho \approx \rho \circ (\mathsf{id} \otimes \rho) \circ \theta$$
$$\Upsilon \circ \langle e, f \rangle \approx \Upsilon \circ \langle f, e \rangle \quad \Upsilon \circ \langle e, e \rangle \approx e \quad \Upsilon \circ \langle e, \varnothing' \rangle \approx e \quad \Upsilon \circ \langle e, \Upsilon \circ \langle f, g \rangle \rangle \approx \Upsilon \circ \langle \Upsilon \circ \langle e, f \rangle, g \rangle$$

where $\theta := \langle \mathsf{fst} \circ \mathsf{fst}, \langle \mathsf{snd} \circ \mathsf{fst}, \mathsf{snd} \rangle \rangle$ and $\varnothing' := \varnothing \circ \mathsf{tt}$. The set-theoretic semantics is extended with:

- $[\![\varnothing_t]\!]$ denotes the function out of $[\![1]\!]$ returning the empty set, $[\![\eta_t]\!]$ denotes the function taking each $x \in [\![t]\!]$ to the singleton set $\{x\}$, and $[\![\Upsilon_t]\!]$ denotes the function taking each pair (x, y) of sets of $[\![t]\!]$ to their union $x \cup y$. These three operations are called the *union presentation of sets*.
- $[\![\mathsf{map}(f)]\!]$ denotes the function taking each set $\{s_1, \ldots, s_n\}$ to the set $\{[\![f]\!](s_1), \ldots, [\![f]\!](s_n)\}$, and $[\![\mu_t]\!]$ denotes the function taking a set of sets of $[\![t]\!]$, say X, to the "n-ary-union" $\bigcup_{x \in X} x$, returning a set of $[\![t]\!]$. Collectively these two operations, and $\varnothing$, form the *set monad with zero*.
- $[\![\rho_{t,t'}]\!]$ denotes the function taking each pair $(x, \{s_1, \ldots, s_n\}) \in [\![t]\!] \times [\![\mathsf{Set}\ t']\!]$ to the set of pairs $\{(x, s_1), \ldots, (x, s_n)\}$. Such an operation is often called a *strength*, and a monad with a strength, such as Set, is called a *strong monad*.

The set monad can be implemented using lists considered up to permutation and duplication as:

$$[\varnothing] := \mathsf{in}' \circ \mathsf{inl} \circ \mathsf{tt} \quad [\eta] := \mathsf{in}' \circ \mathsf{inr} \circ \langle \mathsf{id}, [\varnothing] \circ \mathsf{tt} \rangle \quad [\Upsilon] := \mathsf{fold}(\langle \mathsf{fst} \mid \mathsf{in}' \circ \mathsf{inr} \circ \mathsf{snd} \rangle \circ \delta) \circ \mathsf{swap}$$
$$[\mu_t] := \mathsf{fold}'\langle [\varnothing] \mid [\Upsilon] \rangle \quad [\mathsf{map}_t] := \mathsf{fold}'\langle [\varnothing] \mid \mathsf{in}' \circ \mathsf{inr} \circ \langle f \circ \mathsf{fst}, \mathsf{snd} \rangle \rangle \quad [\rho_t] := \mathsf{fold}(\langle [\varnothing] \circ \mathsf{tt} \mid \mathsf{in}' \circ \mathsf{inr} \circ \omega \rangle \circ \delta)$$

where $\omega := \langle \langle \mathsf{fst}, \mathsf{fst} \circ \mathsf{snd} \rangle, \mathsf{snd} \circ \mathsf{snd} \rangle$ is an isomorphism and so is $\mathsf{swap} := \langle \mathsf{snd}, \mathsf{fst} \rangle$. Although not necessary, we use swap in $[\Upsilon]$ so that it preserves input list order, which makes our proofs easier.

Theorem 2. Let $\triangleright$ denote the function taking lists to sets and *map* denote the map function on lists (i.e., $map(f)(x_1, \ldots, x_n) = (f(x_1), \ldots, f(x_n))$. Then:

$$\triangleright [\![[\varnothing]]\!](x) = [\![\varnothing]\!](x) \quad \triangleright [\![[\eta]]\!](x) = [\![\eta]\!](x) \quad \triangleright [\![[\Upsilon]]\!](x, y) = [\![\Upsilon]\!](\triangleright x, \triangleright y)$$
$$\triangleright [\![[\mu]]\!](x) = [\![\mu]\!](\triangleright map(\triangleright, x)) \quad \triangleright [\![[\mathsf{map}(f)]]\!](x) = [\![\mathsf{map}(f)]\!](\triangleright x) \quad \triangleright [\![[\rho]]\!](x, y) = [\![\rho]\!](x, \triangleright y)$$

Proof. In Coq. □

2.6 Comprehensions as Queries

Although we may use the set monad as described above directly, a comprehension is defined as an expression in the λ-calculus form of CAML that de-sugars into the set monad. Comprehension syntax has many concrete presentations, as well as a convenient normal form, and in this paper we adopt the "do/bind notation" of Haskell [18], and extend the λ-calculus version of CAML with:

$$\frac{}{\Gamma \vdash \varnothing_t : \mathsf{Set}\ t} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathtt{sng}\ e\ :\ \mathsf{Set}\ t} \qquad \frac{\Gamma \vdash e : \mathsf{Set}\ t \qquad \Gamma \vdash e' : \mathsf{Set}\ t}{\Gamma \vdash e \curlyvee e'\ :\ \mathsf{Set}\ t}$$

$$\frac{\Gamma \vdash e : \mathsf{Set}\ t \qquad \Gamma, v : t \vdash f : \mathsf{Set}\ t' \qquad v \notin \Gamma}{\Gamma \vdash \mathtt{do}\ v{:}t \leftarrow e; f : \mathsf{Set}\ t'}$$

and corresponding translations:

$$[\Gamma \vdash \varnothing_t : \mathsf{Set}\ t] := \varnothing_{[t]} \circ \mathsf{tt}_{[\Gamma]} \quad [\Gamma \vdash \mathtt{sng}\ e : \mathsf{Set}\ t] := \eta \circ [\Gamma \vdash e : t] \quad [\Gamma \vdash e \curlyvee e' : \mathsf{Set}\ t] := \curlyvee \circ \langle [\Gamma \vdash e], [\Gamma \vdash e'] \rangle$$

$$[\Gamma \vdash \mathtt{do}\ v{:}t \leftarrow e; f : \mathsf{Set}\ t'] := \mu \circ \mathsf{map}([v{:}t, \Gamma \vdash f]) \circ \rho \circ \langle \mathsf{id}, [\Gamma \vdash e] \rangle$$

Set-theoretically, $\mathtt{do}\ v \leftarrow e; f$ denotes "flat mapping" f through e; i.e., $\cup_{v \in e} f(v)$.

Theorem 3. Let $subst(v \mapsto e, f)$ indicate the capture-avoiding substitution of e for variable v in f and $weaken(v, f)$ the weakening of f to include and ignore fresh variable v. Then:

$$[\![[\Gamma \vdash \mathtt{do}\ v \leftarrow e; \mathtt{sng}\ v]]\!] = [\![[\Gamma \vdash e]]\!] \quad [\![[\Gamma \vdash \mathtt{do}\ v \leftarrow \mathtt{sng}\ e; f]]\!] = [\![[\Gamma \vdash subst(v \mapsto e, f)]]\!]$$

$$[\![[\Gamma \vdash \mathtt{do}\ y \leftarrow (\mathtt{do}\ x \leftarrow m; f)\ ;\ g]]\!] = [\![[\Gamma \vdash \mathtt{do}\ x \leftarrow m\ ;\ (\mathtt{do}\ y \leftarrow f; weaken(x, g))]]\!]$$

$$[\![[\Gamma \vdash \mathtt{do}\ v \leftarrow e; \varnothing]]\!] = [\![[\Gamma \vdash \varnothing]]\!] = [\![[\Gamma \vdash \mathtt{do}\ v \leftarrow \varnothing; f]]\!]$$

Proof. In Coq. □

Additional equations are provable, such as commutativity and idempotency of do/bind, but we single out the equations above because are true in any monad with zero (lists, bags, etc), such as each type in the Boom hierarchy [4], and they can be oriented to form a *strongly normalizing* re-write system [11]. So far, then, our CAML compiler consists of comprehension normalization (representing a simple query optimizer) and then translation into recursion (representing a simple query compiler).

3 Extending CAML with Bi-Algebra ADTs

4 A Standard Library of Bi-Algebra ADTs

5 Application: Generic Compilation of CQL

6 Conclusion

References

[1] Suad Alagić and Philip A. Bernstein. A model theory for generic schema management. In Giorgio Ghelli and Gösta Grahne, editors, *Database Programming Languages*. Springer Berlin Heidelberg, 2002.

[2] Steve Awodey. *Category Theory*. Oxford University Press, Inc., 2nd edition, 2010.

[3] Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, and Erik Paulson. Efficient processing of data warehousing queries in a split execution environment. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 1165–1176, New York, NY, USA, 2011. Association for Computing Machinery.

[4] Alexander Bunkenburg. *The Boom Hierarchy*, pages 1–8. Springer London, London, 1994.

[5] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, page 10–18, New York, NY, USA, 2007. Association for Computing Machinery.

[6] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173 – 202, 1987.

[7] Martin Erwig. Categorical programming with abstract data types. In *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology*, AMAST '98, page 406–421, Berlin, Heidelberg, 1999. Springer-Verlag.

[8] Martin Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11(5):467–492, September 2001.

[9] Martin Erwig. Comprehending adts. *https://www.researchgate.net/publication/228520206_Comprehending_ADTs*, 01 2011.

[10] Marcelo Fiore, Roberto Di Cosmo, and Vincent Balat. Remarks on isomorphisms in typed lambda calculi with empty and sum types. *Annals of Pure and Applied Logic*, 141(1), 2006.

[11] Torsten Grust and Marc H. Scholl. How to comprehend queries functionally. *Journal of Intelligent Information Systems*, 12(2):191–218, Mar 1999.

[12] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: From research prototype to industrial tool. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 805–810, New York, NY, USA, 2005. Association for Computing Machinery.

[13] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Developing metadata-intensive applications with rondo. *Journal of Web Semantics*, 1(1):47 – 74, 2003.

[14] Marko A. Rodriguez. The gremlin graph traversal machine and language (invited talk). *Proceedings of the 15th Symposium on Database Programming Languages - DBPL 2015*, 2015.

[15] Gabriel Scherer. Deciding equivalence with sums and the empty type. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 374–386, New York, NY, USA, 2017. ACM.

[16] Patrick Schultz, David I. Spivak, and Ryan Wisnesky. Algebraic model management: A survey. In Phillip James and Markus Roggenbach, editors, *Recent Trends in Algebraic Development Techniques*. Springer International Publishing, 2017.

[17] Joshua Shinavier and Ryan Wisnesky. Algebraic property graphs, 2019.

[18] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 61–78, New York, NY, USA, 1990. ACM.

[19] Limsoon Wong. *Querying Nested Collections*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1994. AAI9503855.

[20] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08: Eighth Symposium on Operating System Design and Implementation*, December 2008.

Contents