Polymorphic Queries Across Sets, Bags, and Lists*

Limsoon Wong
Real World Computing Partnership Novel Function
Institute of Systems Science Laboratory
Heng Mui Keng Terrace, Singapore 0511
Email: limsoon@iss.nus.sg

22 September 1994

Abstract

Bulk data types such as sets, bags, and lists are an important component of a modern query language. It is desirable that operations on one bulk data type (such as sets) that have analogous counterparts in other bulk data types (such as bags and list) be expressed using the *same* expression. This kind of uniformity is achieved only to a limited degree in object-oriented languages, as these languages have the type information loss problem. Languages based on parametric polymorphism achieve this uniformity in a much better way, but only upto "bracket-switching." That is, in these languages, a query over bags that is analogous to a query over sets, are identical upto interchanging the set and bag brackets. This paper presents a simple extension to a popular parametric polymorphic type system. We show that it is possible to construct a query language around this type system so that the *same* expression can be used to implement analogous queries across sets, bags, and lists. Furthermore, this uniformity is achieved without any loss in type information.

1 Introduction

Sets, bags, and lists are among the most commonly used bulk types. The comprehension syntax [21, 22] has recently been proposed as a syntax for the uniform manipulation of these bulk data types [23, 3, 6, 25]. Using this syntax, it is possible to turn a query on sets into a query on bags or on list by simply switching the set brackets into bag brackets or list brackets. For example,

$$\begin{aligned}
&\mathsf{set\text{-}select} == \lambda\ p.\ \lambda\ S.\ \{\ x \mid x \in S,\ p(x)\ \}\\
&\mathsf{bag\text{-}select} == \lambda\ p.\ \lambda\ S.\ \{\!|\ x \mid x \in S,\ p(x)\ |\!\}\\
&\mathsf{list\text{-}select} == \lambda\ p.\ \lambda\ S.\ [\ x \mid x \in S,\ p(x)\ ]
\end{aligned}$$

defines set-select to be a function that takes a predicate p, a set S, and returns a set containing those elements of S that satisfy p; bag-select to be a function that takes a predicate p, a bag S, and returns a bag containing those

*Appeared in ACM SIGPLAN Notices **30**(4):39-44, April 1995.

elements of S that satisfy p; and list-select to be a function that takes a predicate p, a list S, and returns a list containing those elements of S that satisfy p.

This paper presents an enhancement to this uniformity so that it is possible to write queries that work directly across sets, bags, and lists, without any kind of bracket switching. For example,

$$\mathsf{select} == \lambda\ p.\ \lambda\ S.\ \mathsf{coll}(\ x \mid x \in S,\ p(x))$$

defines select to be a function that takes a predicate p, a collection S, and returns a collection of the *same type* containing those elements of S satisfying p.

The idea behind this enhancement is an ML-like type system that supports polymorphism over collection types. The presence of a type system ensures that whenever a program can be given a type, its execution will be free from runtime type error. An important feature of our system is that it is not necessary to explicitly declare the type of a program. The system is capable of automatically determining if a program can be assigned a type, as well as finding the most general type that can be assigned to it.

Let us first use the four functions above to describe the main kind of collection types used in our system. The function set-select is given the type $(\alpha \to \mathsf{bool}) \to \mathsf{coll}(\mathsf{set}, \alpha) \to \mathsf{coll}(\mathsf{set}, \alpha)$. The function bag-select is given the type $(\alpha \to \mathsf{bool}) \to \mathsf{coll}(\mathsf{bag}, \alpha) \to \mathsf{coll}(\mathsf{bag}, \alpha)$. The function list-select is given the type $(\alpha \to \mathsf{bool}) \to \mathsf{coll}(\mathsf{list}, \alpha) \to \mathsf{coll}(\mathsf{list}, \alpha)$. bool is the type of Boolean values. α is a type variable that ranges over all types for which an equality test is available. $\alpha \to \mathsf{bool}$ is the type of functions taking an object of type α and returning a Boolean value. $\mathsf{coll}(\mathsf{set}, \alpha)$ is the type of sets whose elements are of type α. $\mathsf{coll}(\mathsf{bag}, \alpha)$ is the type of bags whose elements are of type α. $\mathsf{coll}(\mathsf{list}, \alpha)$ is the type of lists whose elements are of type α. Hence set-select works across all kinds of predicates p and sets S, as long as the input type to p agrees with the element type of S. Similarly bag-select works across all kinds of predicates p and bags S and list-select works across all kinds of predicates p and lists S, as long as the input type to p agrees with the element type of S.

However, one cannot apply set-select to a predicate p and a bag S, even when the input type to p is the same as the element type of S, because S is not a set. This is the reason that in previous systems [3, 26] three different expressions are needed to define selection for sets, bags, and lists.

In the system presented here, the function select can be given the type $(\alpha \to \mathsf{bool}) \to \mathsf{coll}(\delta, \alpha) \to \mathsf{coll}(\delta, \alpha)$. Here δ is a type variable that ranges over the symbols set, bag, and list. The type $\mathsf{coll}(\delta, \alpha)$ thus ranges over a δ-collection whose elements are of type α. Hence select works across all kinds of predicates p and collections S of any kind, as long as the input type to p agrees with the element type of S. This idea is fairly straightforward. However, its actual implementation requires some thought. In particular, one has to consider what operations can be made polymorphic across our collection types because not all of them can be generalized.

It is possible to program in C++ a function like select that can be applied to sets, bags, and lists. If we do not use the C++ template mechanism, then the return type of this C++ program, after being applied to p and S, is not always the same type as S. That is, the C++ type system suffers a loss of type information, a phenomenon associated with subtyping [4, 13]. If we do use the template mechanism, then we must give the types explicitly, as well as suffer a potentially large increase in code size when a template is instantiated. Our system does not lose type information and the type of $\mathsf{select}(p)(S)$ is always the *same type* as S. Furthermore, types are always statically inferred and the same piece of codes is used regardless of what select is being applied to.

The remainder of this report is organized as follows. A few further examples are given in Section 2 to illustrate queries that work across sets, bags, and lists. An explicitly-typed calculus is presented in Section 3; it serves as the semantic basis for our language. An implicitly-typed calculus, which is the language used above, is presented in Section 4; we discuss its relationship to the explictly-type calculus and provide a hint on its implementation. Then in Section 5 we present a type inference algorithm for the implicitly-typed calculus.

2 More Examples

In this section, we give four more examples to illustrate queries that are polymorphic across sets, bags, and lists.

2.1 Example: Projection

This first example is the relational projection operation set-proj. This is an operation that extracts certain columns of a relation (equivalently, a set). A similar operation, bag-proj, can be defined on bags. A similar operation, list-proj, can also be defined on lists. The difference between them is that set-proj should removes duplicates from the result, bag-proj should preserve the cardinality, and list-proj should also preserve the ordering or elements. These three functions can be written as shown below. The most general types assigned to these programs are also given; these types are *automatically inferred* by the algorithm in Section 5.

$$\begin{array}{l}
\textsf{set-proj} == \lambda\ p.\ \lambda\ S.\ \{\ p(x)\ |\ x\ \in\ S\ \}\ :\ (\alpha_1 \to \alpha_2) \to \textsf{coll}(\textsf{set}, \alpha_1) \to \textsf{coll}(\textsf{set}, \alpha_2) \\
\textsf{bag-proj} == \lambda\ p.\ \lambda\ S.\ \{\!|\ p(x)\ |\ x\ \in\ S\ |\!\}\ :\ (\alpha_1 \to \alpha_2) \to \textsf{coll}(\textsf{bag}, \alpha_1) \to \textsf{coll}(\textsf{bag}, \alpha_2) \\
\textsf{list-proj} == \lambda\ p.\ \lambda\ S.\ [\ p(x)\ |\ x\ \in\ S\ ]\ :\ (\alpha_1 \to \alpha_2) \to \textsf{coll}(\textsf{list}, \alpha_1) \to \textsf{coll}(\textsf{list}, \alpha_2)
\end{array}$$

For example, let $\#1$ be the function that returns the first component of a pair, then set-proj($\#1$)$\{(1,2), (2,4), (1,3)\}$ returns the set $\{1,2\}$; bag-proj($\#1$)$\{\!|(1,2), (2,4), (1,3)|\!\}$ returns the bag $\{\!|1,1,2|\!\}$; and list-proj($\#1$)$[(1,2), (2,4), (1,3)]$ returns the list $[1,2,1]$.

Observe that the only difference between these programs is that set brackets is used for set-proj, bag brackets for bag-proj, and list brackets for list-proj. Using our system, we can replace all three by the single program below.

$$\textsf{proj} == \lambda\ p.\ \lambda\ S.\ \textsf{coll}(\ p(x)\ |\ x\ \in\ S)\ :\ (\alpha_1 \to \alpha_2) \to \textsf{coll}(\delta, \alpha_1) \to \textsf{coll}(\delta, \alpha_2)$$

Then proj($\#1$) projects the first column of sets, bags, and lists. For example, proj($\#1$)$\{(1,2), (2,4), (1,3)\}$ returns the set $\{1,2\}$; proj($\#1$)$\{\!|(1,2), (2,4), (1,3)|\!\}$ returns the bag $\{\!|1,1,2|\!\}$; and proj($\#1$)$[(1,2), (2,4), (1,3)]$ returns the list $[1,2,1]$.

2.2 Example: Join

The second example is the relational join set-join. This is an operation that pairs records from two sets whenever they agrees on certain columns. There are also similar operations bag-join and list-join on bags of records and lists of records. They can be written as shown below.

$$\begin{array}{l}
\textsf{set-join} == \lambda\ p.\ \lambda\ R.\ \lambda\ S.\ \{\ (x,\ y)\ |\ x\ \in\ R,\ y\ \in\ S,\ p(x,y)\ \} \\
\quad :\ (\alpha_1 \times \alpha_2 \to \textsf{bool}) \to \textsf{coll}(\textsf{set}, \alpha_1) \to \textsf{coll}(\textsf{set}, \alpha_2) \to \textsf{coll}(\textsf{set}, \alpha_1 \times \alpha_2) \\
\textsf{bag-join} == \lambda\ p.\ \lambda\ R.\ \lambda\ S.\ \{\!|\ (x,\ y)\ |\ x\ \in\ R,\ y\ \in\ S,\ p(x,y)\ |\!\} \\
\quad :\ (\alpha_1 \times \alpha_2 \to \textsf{bool}) \to \textsf{coll}(\textsf{bag}, \alpha_1) \to \textsf{coll}(\textsf{bag}, \alpha_2) \to \textsf{coll}(\textsf{bag}, \alpha_1 \times \alpha_2) \\
\textsf{list-join} == \lambda\ p.\ \lambda\ R.\ \lambda\ S.\ [\ (x,\ y)\ |\ x\ \in\ R,\ y\ \in\ S,\ p(x,y)\ ] \\
\quad :\ (\alpha_1 \times \alpha_2 \to \textsf{bool}) \to \textsf{coll}(\textsf{list}, \alpha_1) \to \textsf{coll}(\textsf{list}, \alpha_2) \to \textsf{coll}(\textsf{list}, \alpha_1 \times \alpha_2)
\end{array}$$

These three functions can be replaced by a single program in our calculus, namely:

$$\begin{array}{l}
\textsf{join} == \lambda\ p.\ \lambda\ R.\ \lambda\ S.\ \textsf{coll}(\ (x,\ y)\ |\ x\ \in\ R,\ y\ \in\ S,\ p(x,\ y)\ ) \\
\quad :\ (\alpha_1 \times \alpha_2 \to \textsf{bool}) \to \textsf{coll}(\delta, \alpha_1) \to \textsf{coll}(\delta, \alpha_2) \to \textsf{coll}(\delta, \alpha_1 \times \alpha_2)
\end{array}$$

Let $p : \alpha_1 \times \alpha_2 \rightarrow \mathsf{bool}$ be a join predicate. Then for $R : \mathsf{coll}(\mathsf{set}, \alpha_1)$ and $S : \mathsf{coll}(\mathsf{set}, \alpha_2)$, $\mathsf{join}(p)(R)(S)$ returns the same set of type $\mathsf{coll}(\mathsf{set}, \alpha_1 \times \alpha_2)$ as $\mathsf{set\text{-}join}(p)(R)(S)$. Similarly, for $R : \mathsf{coll}(\mathsf{bag}, \alpha_1)$ and $S : \mathsf{coll}(\mathsf{bag}, \alpha_2)$, $\mathsf{join}(p)(R)(S)$ returns the same bag of type $\mathsf{coll}(\mathsf{bag}, \alpha_1 \times \alpha_2)$ as $\mathsf{bag\text{-}join}(p)(R)(S)$ and for $R : \mathsf{coll}(\mathsf{list}, \alpha_1)$ and $S : \mathsf{coll}(\mathsf{list}, \alpha_2)$, $\mathsf{join}(p)(R)(S)$ returns the same list of type $\mathsf{coll}(\mathsf{list}, \alpha_1 \times \alpha_2)$ as $\mathsf{list\text{-}join}(p)(R)(S)$. However, for $R : \mathsf{coll}(\mathsf{set}, \alpha_1)$ and $S : \mathsf{coll}(\mathsf{bag}, \alpha_2)$ $\mathsf{join}(p)(R)(S)$ does not have a type. The reason is that the inputs R and S are required to be collection types of the same kind. The type inference algorithm given later will statically detect this error.

2.3 Example: Unnesting

The third example is the unnesting operation, set-unnest, of nested relational algebra [20, 17]. The versions for bags and lists are bag-unnest and list-unnest.

$$\begin{array}{l}
\mathsf{set\text{-}unnest} == \lambda\ S.\ \{(x,\ z) \mid (x,\ y) \in S,\ z \in y\} : \mathsf{coll}(\mathsf{set}, \alpha_1 \times \mathsf{coll}(\mathsf{set}, \alpha_2)) \rightarrow \mathsf{coll}(\mathsf{set}, \alpha_1 \times \alpha_2) \\
\mathsf{bag\text{-}unnest} == \lambda\ S.\ \{\!|(x,\ z) \mid (x,\ y) \in S,\ z \in y|\!\} : \mathsf{coll}(\mathsf{bag}, \alpha_1 \times \mathsf{coll}(\mathsf{bag}, \alpha_2)) \rightarrow \mathsf{coll}(\mathsf{bag}, \alpha_1 \times \alpha_2) \\
\mathsf{list\text{-}unnest} == \lambda\ S.\ [(x,\ z) \mid (x,\ y) \in S,\ z \in y] : \mathsf{coll}(\mathsf{list}, \alpha_1 \times \mathsf{coll}(\mathsf{list}, \alpha_2)) \rightarrow \mathsf{coll}(\mathsf{list}, \alpha_1 \times \alpha_2)
\end{array}$$

For example, $\mathsf{set\text{-}unnest}\{(1, \{1, 2\}), (2, \{3\}), (1, \{2\})\}$ returns the set $\{(1, 1), (1, 2), (2, 3)\}$, $\mathsf{bag\text{-}unnest}\{\!|(1,\ \{\!|1, 2|\!\}), (2,\ \{\!|3|\!\}),\ (1,\ \{\!|2|\!\})\}$ returns the bag $\{\!|(1, 1), (1, 2), (1, 2), (2, 3)|\!\}$, and $\mathsf{list\text{-}unnest}[(1, [1, 2]),\ (2, [3]),\ (1, [2])\}$ returns the list $[(1, 1), (1, 2), (2, 3), (1, 2)]$.

These three functions can again be replaced in our calculus by the single program below.

$$\mathsf{unnest} == \lambda\ S.\ \mathsf{coll}(\ (x,\ z) \mid (x,\ y) \in S,\ z \in y\ ) : \mathsf{coll}(\delta, \alpha_1 \times \mathsf{coll}(\delta, \alpha_2)) \rightarrow \mathsf{coll}(\delta, \alpha_1 \times \alpha_2)$$

which behaves exactly as set-unnest for $S : \mathsf{coll}(\mathsf{set}, \alpha_1 \times \mathsf{coll}(\mathsf{set}, \alpha_2))$, as bag-unnest for $S : \mathsf{coll}(\mathsf{bag}, \alpha_1 \times \mathsf{coll}(\mathsf{bag}, \alpha_2))$, and as list-unnest for $S : \mathsf{coll}(\mathsf{list}, \alpha_1 \times \mathsf{coll}(\mathsf{list}, \alpha_2))$.

2.4 Example: Nested Selection

The last example is a selection on nested relation. We list below three different programs for performing selection on different kinds of nested collections.

$$\begin{array}{l}
\mathsf{nest\text{-}select\text{-}a} == \lambda\ S.\ \{\{x \mid x \in y,\ x > 5\} \mid y \in S\} : \mathsf{coll}(\mathsf{set}, \mathsf{coll}(\mathsf{set}, \mathsf{int})) \rightarrow \mathsf{coll}(\mathsf{set}, \mathsf{coll}(\mathsf{set}, \mathsf{int})) \\
\mathsf{nest\text{-}select\text{-}b} == \lambda\ S.\ \{\!|\{x \mid x \in y,\ x > 5\} \mid y \in S|\!\} : \mathsf{coll}(\mathsf{bag}, \mathsf{coll}(\mathsf{set}, \mathsf{int})) \rightarrow \mathsf{coll}(\mathsf{bag}, \mathsf{coll}(\mathsf{set}, \mathsf{int})) \\
\mathsf{nest\text{-}select\text{-}c} == \lambda\ S.\ \{\!|[x \mid x \in y,\ x > 5] \mid y \in S|\!\} : \mathsf{coll}(\mathsf{bag}, \mathsf{coll}(\mathsf{list}, \mathsf{int})) \rightarrow \mathsf{coll}(\mathsf{bag}, \mathsf{coll}(\mathsf{list}, \mathsf{int}))
\end{array}$$

These programs removes integers lesser than 5 from their input, without changing the shape of their input. For example, $\mathsf{nest\text{-}select\text{-}a}\{\{1, 2, 3, 6, 7\}, \{2, 7, 6\}\}$ returns $\{\{6, 7\}\}$, $\mathsf{nest\text{-}select\text{-}b}\{\!|\{1, 2, 3, 6, 7\},\ \{2, 7, 6\}|\!\}$ returns $\{\!|\{6, 7\}, \{6, 7\}|\!\}$, and $\mathsf{nest\text{-}select\text{-}c}\{\!|[1, 2, 3, 6, 7], [2, 7, 6]|\!\}$ returns $\{\!|[6, 7], [7, 6]|\!\}$. Notice that each input involves two kinds of collection types.

These programs and many more that are similar to them can be replaced by a single program using our calculus. Namely:

$$\mathsf{nest\text{-}select} == \lambda\ S.\ \mathsf{coll}(\mathsf{coll}(x \mid x \in y,\ x > 5) \mid y \in S) : \mathsf{coll}(\delta_1, \mathsf{coll}(\delta_2, \mathsf{int})) \rightarrow \mathsf{coll}(\delta_1, \mathsf{coll}(\delta_2, \mathsf{int}))$$

Notice that the type that is assigned (actually inferred by the algorithm to be presented later) contains two type variables δ_1 and δ_2 which correctly captures the two kinds of collection types that might be involved.

In the remainder of this report, we will develop the system that allows us to write programs such as select, proj, join, unnest, and nest-select which works polymorphically across collection types.

3 Explicitly-Typed Calculus

As a basis for our language, we develop the explicitly-typed calculus in this section. This calculus is based on the formalism of [2, 12, 26], which is recently being used as a query language for complex heterogenous genomic data sources [9]. The fragment dealing with sets is known [26] to be equivalent to several nested relational algebras [20, 17, 5, 15].

The types used in the explicitly-typed calculus are either eq-types t or noneq-types T. The eq-types are those types where equality tests can be performed on their objects; the noneq-types are those types where equality tests are not available. This distinction is necessary because we require that elements of sets, bags, and lists to have equality tests. The grammars for our types are given below.

$$t ::= b \mid t_1 \times t_2 \mid \mathsf{coll}(\mathsf{set}, t) \mid \mathsf{coll}(\mathsf{bag}, t) \mid \mathsf{coll}(\mathsf{list}, t)$$
$$T ::= b \mid T_1 \times T_2 \mid \mathsf{coll}(\mathsf{set}, t) \mid \mathsf{coll}(\mathsf{bag}, t) \mid \mathsf{coll}(\mathsf{list}, t) \mid T_1 \to T_2$$

Here, b ranges over base types such as bool and int. The objects of type $\mathsf{coll}(\mathsf{set}, t)$ are finite sets whose elements are of type t. The objects of type $\mathsf{coll}(\mathsf{bag}, t)$ are finite multisets (also called bags) whose elements are of type t. The objects of type $\mathsf{coll}(\mathsf{list}, t)$ are finite lists whose elements are of type t. The objects of type $T_1 \to T_2$ are functions whose inputs are objects of type T_1 and whose outputs are objects of type T_2. The objects of type $T_1 \times T_2$ are pairs whose first components are objects of type T_1 and whose second components are objects of type T_2. We use the notation $\{o_1, ..., o_n\}$ for sets, $\{\!|o_1, ..., o_n|\!\}$ for bags, and $[o_1, ..., o_n]$ for lists. As usual, sets are identified upto commutativity and idempotence and bags are identified upto commutativity.

The expressions of the explicitly-typed calculus are given by a set of typing rules of the form $\Gamma \vdash e : T$, saying that the expression e is assigned type T in the environment Γ. An environment is just a list $x_1 : T_1, ..., x_n : T_n$ with $x_1, ..., x_n$ distinct. We write $\Gamma, x : T$ for the environment obtained by adding $x : T$ to Γ. The Barendregt convention on bound variables [1] is adopted. That is, expressions are identified upto renaming of bound variables. For example, $\lambda x^T.x$ is identified with $\lambda y^T.y$.

The expressions are basically those of the simply-typed lambda calculus [8]:

$$\frac{x : T \text{ is in } \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma,\ x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x^{T_1}.e : T_1 \to T_2} \qquad \frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1 e_2 : T_2}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 \times T_2} \qquad \frac{}{\Gamma \vdash \#1 : T_1 \times T_2 \to T_1} \qquad \frac{}{\Gamma \vdash \#2 : T_1 \times T_2 \to T_2}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T}$$

augmented with constants (such as the integers), primitive functions (such as equality tests), and the following operators on collection types:

$$\frac{}{\Gamma \vdash \mathsf{empty}^{\mathsf{set},t} : \mathsf{coll}(\mathsf{set}, t)} \quad \frac{}{\Gamma \vdash \mathsf{empty}^{\mathsf{bag},t} : \mathsf{coll}(\mathsf{bag}, t)} \quad \frac{}{\Gamma \vdash \mathsf{empty}^{\mathsf{list},t} : \mathsf{coll}(\mathsf{list}, t)}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathsf{sng}^{\mathsf{set}}(e) : \mathsf{coll}(\mathsf{set}, t)} \quad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathsf{sng}^{\mathsf{bag}}(e) : \mathsf{coll}(\mathsf{bag}, t)} \quad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathsf{sng}^{\mathsf{list}}(e) : \mathsf{coll}(\mathsf{list}, t)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{coll}(\mathsf{set}, t) \quad \Gamma \vdash e_2 : \mathsf{coll}(\mathsf{set}, t)}{\Gamma \vdash e_1 \cup e_2 : \mathsf{coll}(\mathsf{set}, t)} \quad \frac{\Gamma, x : t_2 :\vdash e_1 : \mathsf{coll}(\mathsf{set}, t_1) \quad \Gamma \vdash e_2 : \mathsf{coll}(\mathsf{set}, t_2)}{\Gamma \vdash \mathsf{coll}^{\mathsf{set}}(e_1 \mid x^{t_2} \in e_2) : \mathsf{coll}(\mathsf{set}, t_1)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{coll}(\mathsf{bag}, t) \quad \Gamma \vdash e_2 : \mathsf{coll}(\mathsf{bag}, t)}{\Gamma \vdash e_1 \uplus e_2 : \mathsf{coll}(\mathsf{bag}, t)} \quad \frac{\Gamma, x : t_2 \vdash e_1 : \mathsf{coll}(\mathsf{bag}, t_1) \quad \Gamma \vdash e_2 : \mathsf{coll}(\mathsf{bag}, t_2)}{\Gamma \vdash \mathsf{coll}^{\mathsf{bag}}(e_1 \mid x^{t_2} \in e_2) : \mathsf{coll}(\mathsf{bag}, t_1)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{coll}(\mathsf{list}, t) \quad \Gamma \vdash e_2 : \mathsf{coll}(\mathsf{list}, t)}{\Gamma \vdash e_1 @ e_2 : \mathsf{coll}(\mathsf{list}, t)} \quad \frac{\Gamma, x : t_2 \vdash e_1 : \mathsf{coll}(\mathsf{list}, t_1) \quad \Gamma \vdash e_2 : \mathsf{coll}(\mathsf{list}, t_2)}{\Gamma \vdash \mathsf{coll}^{\mathsf{list}}(e_1 \mid x^{t_2} \in e_2) : \mathsf{coll}(\mathsf{list}, t_1)}$$

It should be remarked that every well-typed expression of the explictly-typed calculus comes from a *unique* typing derivation according to the above rules.

The meanings for the first group of operators are the usual ones of the lambda calculus; see [8]. We give a more detailed description of the second group, which is based on monads [2, 12]. The meaning of $\mathsf{empty}^{\mathsf{set},t}$ is the empty set of type t. The meaning of $\mathsf{sng}^{\mathsf{set}}(e)$ is the singleton set containg the object e. The meaning of $e_1 \cup e_2$ is the union of sets e_1 and e_2. The meaning of $\mathsf{coll}^{\mathsf{set}}(e_1 \mid x^T \in e_2)$ is the set obtained by flatmapping the function λx^T over the set e_2. That is, if e_2 is the set $\{o_1, ..., o_n\}$ and $\lambda x^T.e_1$ is the function f, then $\mathsf{coll}^{\mathsf{set}}(e_1 \mid x^T \in e_2)$ is the set $f(o_1) \cup \ldots \cup f(o_n)$.

The meaning of $\mathsf{empty}^{\mathsf{bag},t}$ is the empty bag of type t. The meaning of $\mathsf{sng}^{\mathsf{bag}}(e)$ is the singleton bag containing the object e. The meaning of $e_1 \uplus e_2$ is the additive union of bags e_1 and e_2. For example, $\{\!|1, 1, 2|\!\} \uplus \{\!|2, 3, 1, 4|\!\}$ is $\{\!|1, 1, 1, 2, 2, 3, 4|\!\}$. The meaning of $\mathsf{coll}^{\mathsf{bag}}(e_1 \mid x^T \in e_2)$ is the bag obtained by flatmapping the function λx^T over the bag e_2. That is, if e_2 is the bag $\{\!|o_1, ..., o_n|\!\}$ and $\lambda x^T.e_1$ is the function f, then $\mathsf{coll}^{\mathsf{bag}}(e_1 \mid x^T \in e_2)$ is the bag $f(o_1) \uplus \ldots \uplus f(o_n)$.

The meaning of $\mathsf{empty}^{\mathsf{list},t}$ is the empty list of type t. The meaning of $\mathsf{sng}^{\mathsf{list}}(e)$ is the singleton list containing the object e. The meaning of $e_1 @ e_2$ is the concatenation of list e_1 to e_2. The meaning of $\mathsf{coll}^{\mathsf{list}}(e_1 \mid x^T \in e_2)$ is the list obtained by flatmapping the function λx^T over the list e_2. That is, if e_2 is the list $[o_1, ..., o_n]$ and $\lambda x^T.e_1$ is the function f, then $\mathsf{coll}^{\mathsf{list}}(e_1 \mid x^T \in e_2)$ is the list $f(o_1) @ \ldots @ f(o_n)$.

The expressions of the explicitly-typed calculus are evaluated according to a set of rules of the form $e_1 : T \leadsto e_2$, which says that $\vdash e_1 : T_1$ and e_1 rewrites in one step to e_2. The set of rules we have implements a deterministic call-by-name semantics. (See [10, 7] for various ways of specifying operational semantics.) A call-by-value operational semantics can also be given; see [18, 19]. We list the rules for evaluating $\mathsf{coll}^{\mathsf{set}}$, $\mathsf{coll}^{\mathsf{bag}}$, and $\mathsf{coll}^{\mathsf{list}}$ below and omit the rest.

$$\mathsf{coll}^{\mathsf{set}}(e_1 \mid x^{t_2} \in \mathsf{empty}^{\mathsf{set},t_2}) : \mathsf{coll}(\mathsf{set}, t_1) \leadsto \mathsf{empty}^{\mathsf{set},t_1}$$

$$\mathsf{coll}^{\mathsf{bag}}(e_1 \mid x^{t_2} \in \mathsf{empty}^{\mathsf{bag},t_2}) : \mathsf{coll}(\mathsf{bag}, t_1) \leadsto \mathsf{empty}^{\mathsf{bag},t_1}$$

$$\mathsf{coll}^{\mathsf{list}}(e_1 \mid x^{t_2} \in \mathsf{empty}^{\mathsf{list},t_2}) : \mathsf{coll}(\mathsf{list}, t_1) \leadsto \mathsf{empty}^{\mathsf{list},t_1}$$

$$\mathsf{coll}^{\mathsf{set}}(e_1 \mid x^{t_2} \in \mathsf{sng}^{\mathsf{set}}(e_2)) : \mathsf{coll}(\mathsf{set}, t_1) \leadsto e_1[e_2/x]$$

$$\mathsf{coll}^{\mathsf{bag}}(e_1 \mid x^{t_2} \in \mathsf{sng}^{\mathsf{bag}}(e_2)) : \mathsf{coll}(\mathsf{bag}, t_1) \leadsto e_1[e_2/x]$$

$$\mathsf{coll}^{\mathsf{list}}(e_1 \mid x^{t_2} \in \mathsf{sng}^{\mathsf{list}}(e_2)) : \mathsf{coll}(\mathsf{list}, t_1) \leadsto e_1[e_2/x]$$

$$\mathsf{coll}^{\mathsf{set}}(e_1 \mid x^{t_2} \in (e_2 \cup e_3)) : \mathsf{coll}(\mathsf{set}, t_1) \rightsquigarrow \mathsf{coll}^{\mathsf{set}}(e_1 \mid x^{t_2} \in e_2) \cup \mathsf{coll}^{\mathsf{set}}(e_1 \mid x^{t_2} \in e_3)$$

$$\mathsf{coll}^{\mathsf{bag}}(e_1 \mid x^{t_2} \in (e_2 \uplus e_3)) : \mathsf{coll}(\mathsf{bag}, t_1) \rightsquigarrow \mathsf{coll}^{\mathsf{bag}}(e_1 \mid x^{t_2} \in e_2) \uplus \mathsf{coll}^{\mathsf{bag}}(e_1 \mid x^{t_2} \in e_3)$$

$$\mathsf{coll}^{\mathsf{list}}(e_1 \mid x^{t_2} \in (e_2 @ e_3)) : \mathsf{coll}(\mathsf{list}, t_1) \rightsquigarrow \mathsf{coll}^{\mathsf{list}}(e_1 \mid x^{t_2} \in e_2) @ \mathsf{coll}^{\mathsf{list}}(e_1 \mid x^{t_2} \in e_3)$$

These operational rules are type safe and sound. That is, if e_1 evaluates to e_2 then e_1 and e_2 have the same type and same meaning.

Theorem 3.1 *Let $e_1 : T \rightsquigarrow e_2$. Then $\vdash e_2 : T$ and $e_1 = e_2$.* □

Recall that in Section 2, we have used the comprehension syntax $\{e \mid \Delta\}$, $\{\!|e \mid \Delta|\!\}$, and $[e \mid \Delta]$, where Δ has the form A_1, ..., A_n and each A_i is either of the form $x_i \in e_i$ or is an expression e_i of Boolean type. These can be treated as syntactic sugars and can be eliminated a-la Wadler [23]. Namely, recursively translate $\{e \mid x_1^{t_1} \in e_1, \Delta\} : \mathsf{coll}(\mathsf{set}, t)$ as $\mathsf{coll}^{\mathsf{set}}(\{e \mid \Delta\} \mid x_1^{t_1} \in e_1)$; $\{e \mid e_1, \Delta\} : \mathsf{coll}(\mathsf{set}, t)$ as $\mathsf{if}\ e_1\ \mathsf{then}\ \{e \mid \Delta\}\ \mathsf{else}\ \mathsf{empty}^{\mathsf{set},t}$; and $\{e \mid \} : \mathsf{coll}(\mathsf{set}, t)$ as $\mathsf{sng}^{\mathsf{set}}(e)$. The translations of bag-comprehension and list-comprehension are analogous.

4 Implicitly-Typed Calculus

The expressions of the explicitly-typed calculus contains type tags, which are indicated by superscripts in constructs such as $\mathsf{empty}^{\mathsf{set},t}$, $\mathsf{sng}^{\mathsf{bag}}(e)$, $\mathsf{coll}^{\mathsf{list}}(e_1 \mid x^t \in e_2)$, $\lambda x^T.e$, etc. In this section, we described the implicitly-typed calculus and relate it to the explicitly-typed calculus.

The types used in the implicitly-typed calculus are same as those used in the explicitly typed calculus. The expressions of the implicitly-typed calculus are same as those of the explicitly-typed calculus but without any type tags. The typing rules for the implicitly-typed calculus are the same as the explicitly-typed rules after the type tags are stripped.

We list only the rules for sets, bags, and lists below. We use D as a meta symbol for set, bag, and list.

$$\frac{}{\Gamma \vdash \mathsf{empty} : \mathsf{coll}(D, t)} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathsf{sng}(e) : \mathsf{coll}(D, t)} \qquad \frac{\Gamma, x : t_2 \vdash e_1 : \mathsf{coll}(D, t_1) \quad \Gamma \vdash e_2 : \mathsf{coll}(D, t_2)}{\Gamma \vdash \mathsf{coll}(e_1 \mid x \in e_2) : \mathsf{coll}(D, t_1)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{coll}(\mathsf{set}, t) \quad \Gamma \vdash e_2 : \mathsf{coll}(\mathsf{set}, t)}{\Gamma \vdash e_1 \cup e_2 : \mathsf{coll}(\mathsf{set}, t)} \qquad \frac{\Gamma \vdash e_1 : \mathsf{coll}(\mathsf{bag}, t) \quad \Gamma \vdash e_2 : \mathsf{coll}(\mathsf{bag}, t)}{\Gamma \vdash e_1 \uplus e_2 : \mathsf{coll}(\mathsf{bag}, t)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{coll}(\mathsf{list}, t) \quad \Gamma \vdash e_2 : \mathsf{coll}(\mathsf{list}, t)}{\Gamma \vdash e_1 @ e_2 : \mathsf{coll}(\mathsf{list}, t)}$$

It is important to observe that, in contrast to the explicitly-typed calculus, expressions in the implicitly-typed calculus can be given many distinct types. However, the two systems are closely related. Let us write $e \equiv \mathit{erase}(e')$ to mean that e is an expression of the implicitly-typed calculus that is obtained by deleting the type tags from the explicitly-typed expression e'. For example, $\mathsf{coll}(x \mid x \in \mathsf{sng}(\mathsf{sng}(1))) \equiv \mathit{erase}(\mathsf{coll}^{\mathsf{set}}(x \mid x^{\mathsf{coll}(\mathsf{set},\mathsf{int})} \in \mathsf{sng}^{\mathsf{set}}(\mathsf{sng}^{\mathsf{set}}(1))))$. Then

Theorem 4.1
- *Let $\Gamma \vdash e : T$ in the explicitly-typed calculus. Then there is e' such that $e' \equiv \mathrm{erase}(e)$ and $\Gamma \vdash e' : T$ in the implicitly-typed calculus.*
- *Let $\Gamma \vdash e : T$ in the implicitly-typed calculus. Then there is e' such that $e \equiv \mathrm{erase}(e')$ and $\Gamma \vdash e' : T$ in the explicitly-typed calculus.* □

This theorem says that an implicitly-typed expression is well typed iff it can be obtained by erasing the type tags of an explicitly-typed expression that is well typed. Naturally, several distinct explicitly-typed expressions can be erased into the same implicitly-typed expression. This provides a basis for giving semantics to the implicitly-typed calculus: The meaning of an implicitly-typed expression $\Gamma \vdash e : T$ is simply the set of explicitly-typed expressions e' such that $e \equiv \mathit{erase}(e')$ and $\Gamma \vdash e' : T$. In other words, e is just a shorthand for each of the e'.

We now give an operational semantics for directly evaluating the implicitly-typed calculus. We list only the rewrite rules for collection types. Let $e_1 \leadsto e_2$ means e_1 evaluates to e_2 in one step in the implicitly-typed calculus.

$$\mathsf{coll}(e_1 \mid x \in \mathsf{empty}) \leadsto \mathsf{empty}$$

$$\mathsf{coll}(e_1 \mid x \in \mathsf{sng}(e_2)) \leadsto e_1[e_2/x]$$

$$\mathsf{coll}(e_1 \mid x \in (e_2 \cup e_3)) \leadsto \mathsf{coll}(e_1 \mid x \in e_2) \cup \mathsf{coll}(e_1 \mid x \in e_3)$$

$$\mathsf{coll}(e_1 \mid x \in (e_2 \uplus e_3)) \leadsto \mathsf{coll}(e_1 \mid x \in e_2) \uplus \mathsf{coll}(e_1 \mid x \in e_3)$$

$$\mathsf{coll}(e_1 \mid x \in (e_2 @ e_3)) \leadsto \mathsf{coll}(e_1 \mid x \in e_2) @ \mathsf{coll}(e_1 \mid x \in e_3)$$

Notice that the last three rules express a dynamic test on their input in order to decide whether $\cup$, $\uplus$, or @ are to be used to combine the results from e_2 and e_3. In this way, the single coll operator is used to simulates $\mathsf{coll}^{\mathsf{set}}$, $\mathsf{coll}^{\mathsf{bag}}$, and $\mathsf{coll}^{\mathsf{list}}$.

These rules are sound and type safe. That is, it takes well-typed expressions into well-typed expressions. Moreover, evaluation preserves meanings of expressions.

Theorem 4.2 *Suppose $\vdash e_1 : T$ and $e_1 \leadsto e_2$ in the implicitly-typed calculus. Then there are e_1' and e_2' such that $e_1' : T \leadsto e_2'$ in the explicitly-typed calculus and $e_1 \equiv \mathit{erase}(e_1')$ and $e_2 \equiv \mathit{erase}(e_2')$.* □

Recall that in Section 2, we have used the polymorphic comprehension syntax $\mathsf{coll}(e \mid \Delta)$ to simultaneously implement $\{e \mid \Delta\}$, $\{\!|e \mid \Delta|\!\}$, and $[e \mid \Delta]$. We show how it can be treated as a syntactic sugar in our implicitly-typed calculus using the technique of Wadler [23]. Namely, recursively translate $\mathsf{coll}(e \mid x \in e_1, \Delta)$ as $\mathsf{coll}(\mathsf{coll}(e \mid \Delta) \mid x \in e_1)$; $\mathsf{coll}(e \mid e_1, \Delta)$ as $\mathsf{if}\ e_1\ \mathsf{then}\ \mathsf{coll}(e \mid \Delta)\ \mathsf{else}\ \mathsf{empty}$; and $\mathsf{coll}(e \mid\ )$ as $\mathsf{sng}(e)$. This implementation of polymorphic comprehension explains why we have empty, sng, and coll in the implicitly-typed calculus.

It is tempting to add an operator $\bullet$ to simultaneously implement $\cup$, $\uplus$, and @ in the implicitly-typed calculus. However, there is no way to add such a $\bullet$ into the implicitly-typed calculus.

Theorem 4.3 *The $\cup$, $\uplus$, and @ constructs of the implicitly-typed calculus cannot be replaced by a single operator $\bullet : \mathsf{coll}(\delta, \alpha) \to \mathsf{coll}(\delta, \alpha)$.*

Proof. Consider the function, in the explicitly-typed calculus, $P(x) = \mathsf{if}\ x = \mathsf{empty}^{\mathsf{set},\mathsf{int}}\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ x = (\mathsf{sng}^{\mathsf{set}}(1) \cup \mathsf{sng}^{\mathsf{set}}(2))$. Let O_p be $(\mathsf{sng}^{\mathsf{set}}(1) \cup \mathsf{sng}^{\mathsf{set}}(1) \cup \mathsf{sng}^{\mathsf{set}}(2))$. Then $P(O_P)$ returns true. Also consider the function, in the explicitly-typed calculus, $Q(x) = \mathsf{if}\ x = \mathsf{empty}^{\mathsf{bag},\mathsf{int}}\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ x = (\mathsf{sng}^{\mathsf{bag}}(1) \uplus \mathsf{sng}^{\mathsf{bag}}(2))$. Let O_Q be $(\mathsf{sng}^{\mathsf{bag}}(1) \uplus \mathsf{sng}^{\mathsf{bag}}(1) \uplus \mathsf{sng}^{\mathsf{bag}}(2))$. Then $Q(O_Q)$ returns false. If we replace $\cup$, $\uplus$, and @ by $\bullet$ in the implicitly-typed calculus, then both P and Q will get mapped to the same expression $R(x) = \mathsf{if}\ x = \mathsf{empty}\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ x = (\mathsf{sng}(1) \bullet \mathsf{sng}(2))$. Similarly, both O_P and O_Q will get mapped to the same object $O = (\mathsf{sng}(1) \bullet \mathsf{sng}(1) \bullet \mathsf{sng}(2))$. But then what should $R(O)$ return? It cannot return true without violating Theorem 4.2 because $Q(O_Q)$ returns false. It cannot return false without violating Theorem 4.2 because $P(O_P)$ returns true. Hence $\bullet$ cannot be admitted into the calculus without destroying Theorem 4.2, which is the basis of semantic correctness for the implicitly-typed calculus. □

5 Type Inference Algorithm

In the previous section we have given the semantics of the implicitly-typed calculus and the basic technique for evaluating queries written in it. However, the evaluation is only correct for well-typed expressions. In this section, we show that we can decide whether an expression is well typed or not at compile time. The algorithm is a type-inference algorithm, which returns a principal typing scheme if the expression is well typed and fails otherwise.

We need the notion of nonground types in order to explain the notion of principal typing scheme. There are two kinds of nonground types: those having an equality test are denoted by the meta symbol τ, those having no equality test are denoted by the meta symbol σ. The grammar is given below.

$$\tau ::= \alpha \mid b \mid \tau_1 \times \tau_2 \mid \mathsf{coll}(\mathsf{set}, \tau) \mid \mathsf{coll}(\mathsf{bag}, \tau) \mid \mathsf{coll}(\mathsf{list}, \tau) \mid \mathsf{coll}(\delta, \tau)$$
$$\sigma ::= \alpha \mid b \mid \sigma_1 \times \sigma_2 \mid \mathsf{coll}(\mathsf{set}, \tau) \mid \mathsf{coll}(\mathsf{bag}, \tau) \mid \mathsf{coll}(\mathsf{list}, \tau) \mid \mathsf{coll}(\delta, \tau) \mid \beta \mid \sigma_1 \to \sigma_2$$

They respectively extends the eq-types and noneq-types given in Section 3 with type variables α, β, and δ. Type variables denoted by α can only be instantiated with types having equality tests. Type variables denoted by β can be instantiated with all types. Type variables denoted by δ can only be instantiated with the symbols set, bag, and list. A type is ground is it contains no type variable.

We use θ to denote a substitution from type variable to types of the appropriate kind. We write $\sigma\theta$ for the type obtained by applying the substitution θ to the type σ. A type σ_1 is more general than a type σ_2 if there is a substitution θ such that $\sigma_1\theta = \sigma_2$. A substitution θ is ground for σ if $\sigma\theta$ is a ground.

We extend the notion of typing $\Gamma \vdash e : T$ introduced in Section 3 to nonground types, by allowing type variables to appear in Γ and T. Then $\Gamma \vdash e : \sigma$ is a typing scheme for the implictly-typed expression e if there is a ground substitution θ such that $\Gamma\theta \vdash e : \sigma\theta$ is a typing in the implicitly-typed calculus. A typing scheme $\Gamma_1 \vdash e : \sigma_1$ is principal if for every other typing scheme $\Gamma_2 \vdash e : \sigma_2$, there is a substitution θ so that $\sigma_1\theta = \sigma_2$ and for every $x : \sigma_1$ in Γ_1, $x : \sigma_1\theta$ is in Γ_2.

Our type inference algorithm contains two parts. The first part is for generating a list of type constraints that must be satisfied for an expression to be well typed. The second part is for checking the satisfiability of these type constraints.

A type constraint is pair $(\sigma_1 \doteq \sigma_2)$. We use $\mathcal{E}$ to denote a list of type constraints. We write $\mathcal{E}_1 + \mathcal{E}_2$ to denote the concatenation of two such lists. We write $\mathcal{E} + (\sigma_1 \doteq \sigma_2)$ to denote adding $(\sigma_1 \doteq \sigma_2)$ to $\mathcal{E}$. A list of type constraints $\mathcal{E}$ is satisfiable by a substitution θ if for each $(\sigma_1 \doteq \sigma_2)$ in $\mathcal{E}$, it is the case that $\sigma_1\theta = \sigma_2\theta$; that is, θ is a unifier of every type constraint in $\mathcal{E}$.

The first part of our algorithm is defined by a set of rules of the form $\Gamma \vdash e : \beta \triangleright \mathcal{E}$. These rules define a procedure that given Γ, e, and β returns $\mathcal{E}$ that is satisfiable iff e is well typed. We list only the rules for empty, sng, coll, $\cup$, $\uplus$, and @ below.

$$\frac{\delta \text{ and } \alpha \text{ are fresh.}}{\Gamma \vdash \mathsf{empty} : \beta \triangleright (\beta \doteq \mathsf{coll}(\delta, \alpha))} \qquad \frac{\Gamma \vdash e : \alpha \triangleright \mathcal{E} \quad \delta \text{ and } \alpha \text{ are fresh.}}{\Gamma \vdash \mathsf{sng}(e) : \beta \triangleright \mathcal{E} + (\beta \doteq \mathsf{coll}(\delta, \alpha))}$$

$$\frac{\Gamma, x : \alpha_2 \vdash e_1 : \beta \triangleright \mathcal{E}_1 \quad \Gamma \vdash e_2 : \beta_2 \triangleright \mathcal{E}_2 \quad \alpha,\ \alpha_2, \beta_2, \text{ and } \delta \text{ are fresh.}}{\Gamma \vdash \mathsf{coll}(e_1 \mid x \in e_2) : \beta \triangleright \mathcal{E}_1 + \mathcal{E}_2 + (\beta_2 \doteq \mathsf{coll}(\delta, \alpha_2)) + (\beta \doteq \mathsf{coll}(\delta, \alpha))}$$

$$\frac{\Gamma \vdash e_1 : \beta \triangleright \mathcal{E}_1 \quad \Gamma \vdash e_2 : \beta \triangleright \mathcal{E}_2 \quad \alpha \text{ is fresh.}}{\Gamma \vdash e_1 \cup e_2 : \beta \triangleright \mathcal{E}_1 + \mathcal{E}_2 + (\beta \doteq \mathsf{coll}(\mathsf{set}, \alpha))} \qquad \frac{\Gamma \vdash e_1 : \beta \triangleright \mathcal{E}_1 \quad \Gamma \vdash e_2 : \beta \triangleright \mathcal{E}_2 \quad \alpha \text{ is fresh.}}{\Gamma \vdash e_1 \uplus e_2 : \beta \triangleright \mathcal{E}_1 + \mathcal{E}_2 + (\beta \doteq \mathsf{coll}(\mathsf{bag}, \alpha))}$$

$$\frac{\Gamma \vdash e_1 : \beta \triangleright \mathcal{E}_1 \quad \Gamma \vdash e_2 : \beta \triangleright \mathcal{E}_2 \qquad \alpha \text{ is fresh.}}{\Gamma \vdash e_1 @ e_2 : \beta \triangleright \mathcal{E}_1 + \mathcal{E}_2 + (\beta \doteq \mathsf{coll}(\mathsf{list}, \alpha))}$$

We can prove that these rules indeed produce typing schemes.

Proposition 5.1

- *Let $\Gamma \vdash e : T$. Then there are β and $\mathcal{E}$ such that $\Gamma \vdash e : \beta \triangleright \mathcal{E}$ and $\mathcal{E}$ is satisfiable.*
- *Let $\Gamma \vdash e : \beta \triangleright \mathcal{E}$. Suppose $\mathcal{E}$ is satisfied by a ground substitution θ. Then $\Gamma\theta \vdash e : \beta\theta$.* □

The second part of our algorithm is a type unification algorithm $\mathcal{M}$. To simplify the presentation, $\mathcal{E}$ is extended slightly include constraints of the form $(D_1 \doteq D_2)$ where D_i range over set, bag, list, and δ. The rules are given below, where symmetric cases have been omitted. They are essentially the usual type unification rules, except the first two which are added to deal with polymorphic collection types.

$\mathcal{M}(\mathcal{E} + (\mathsf{coll}(D_1, \tau_1) \doteq \mathsf{coll}(D_2, \tau_2))) = \mathcal{M}(\mathcal{E} + (D_1 \doteq D_2) + (\tau_1 \doteq \tau_2))$.

$\mathcal{M}(\mathcal{E} + (\delta \doteq D)) = (\mathcal{M}(\mathcal{E}\theta)) \circ \theta$, where $\theta(\delta) = D$ and $\theta(x) = x$ otherwise.

$\mathcal{M}() = id$, where id is the identity substitution.

$\mathcal{M}(\mathcal{E} + (\sigma \doteq \sigma)) = \mathcal{M}(\mathcal{E})$

$\mathcal{M}(\mathcal{E} + (\beta \doteq \sigma)) = (\mathcal{M}(\mathcal{E}\theta)) \circ \theta$, where σ is not β and β does not occur in σ and $\theta(\alpha) = \sigma$ and $\theta(x) = x$ otherwise.

$\mathcal{M}(\mathcal{E} + (\sigma_1 \to \sigma_2 \doteq \sigma_3 \to \sigma_4)) = \mathcal{M}(\mathcal{E} + (\sigma_1 \doteq \sigma_3) + (\sigma_2 \doteq \sigma_4))$.

$\mathcal{M}(\mathcal{E}((\sigma_1 \times \sigma_2 \doteq \sigma_3 \times \sigma_4)) = \mathcal{M}(\mathcal{E} + (\sigma_1 \doteq \sigma_3) + (\sigma_2 \doteq \sigma_4))$.

$\mathcal{M}(\mathcal{E} + (\alpha \doteq \tau)) = (\mathcal{M}(\mathcal{E}\theta)) \circ \theta$, where τ is not α and α does not occur in τ and $\theta(\alpha) = \tau$ and $\theta(x) = x$ otherwise.

$\mathcal{M}(\mathcal{E} + (\alpha \doteq \sigma_1 \times \sigma_2)) = \mathcal{M}(\mathcal{E} + (\alpha \doteq \alpha_1 \times \alpha_2) + (\alpha_1 \doteq \sigma_1) + (\alpha_2 \doteq \sigma_2))$, where α_1 and α_2 are fresh.

We can show that

Proposition 5.2

- *Suppose $\mathcal{E}$ is satisfiable. Then $\mathcal{M}(\mathcal{E})$ is its most general unifier.*
- *Suppose $\mathcal{E}$ is not satisfiable. Then $\mathcal{M}(\mathcal{E})$ fails in finite time.* □

Combining Propositions 5.2 and 5.1, we have the desired type inference procedure.

Theorem 5.3 *There is an algorithm such that for any closed expression e of the implicitly-typed calculus: if e can be assigned a type, then the algorithm returns the principal typing scheme for e; or if e cannot be assigned a type, then the algorithm fails in finite time.* □

6 Remarks

We summarize the contributions of this paper. The idea of polymorphic collection types is presented via examples in Section 2. This idea is realized in the implicitly-typed calculus of Section 4. This calculus is given a rigorous semantics by showing that its expressions can be treated as shorthands for the explicitly-typed calculus of Section 3. Furthermore, the execution of well-typed queries in implictly-typed calculus is shown to be free from runtime type error. In addition, we show that certain operators such as $\cup$, $\uplus$, and @ cannot be made polymorphic across collection types. Finally, a simple type inference algorithm is given in Section 5 to decide at compile time if an implicitly-typed expression is well typed.

For simplicity of exposition, we used only pairs rather than general records in this paper. However, there is no difficulty in introducing the latter. In fact, our proposal for polymorphic collection types is readily combined with many type systems that have polymorphic records [14, 11, 16, 24].

References

[1] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and Foundations of Mathematics*. Elsevier, 1984.

[2] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *LNCS 646: Proceedings of 4th International Conference on Database Theory*, Berlin, Germany, October 1992, pages 140–154.

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

[4] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Survey*, 17(4):471–522, December 1985.

[5] L. S. Colby. A recursive algebra for nested relations. *Information Systems*, 15(5):567–582, 1990.

[6] L. Fegaras. A uniform calculus for bulk data types, February 1994. Manuscript available from `fegaras@cse.ogi.edu`.

[7] C. A. Gunter. Forms of semantic specification. *Bulletin of the European Association for Theoretical Computer Science*, 45:98–113, 1991.

[8] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[9] K. Hart, L. Wong, C. Overton, and P. Buneman. Using a query language to integrate biological data. Talk given at the Meeting on the Interconnection of Molecular Biology Databases, Stanford, August 1994.

[10] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics*. John Wiley and Sons, Chichester, England, 1990.

[11] L. A. Jategaonkar and J. C. Mitchell. ML with extended pattern matching and subtypes. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[12] L. Libkin and L. Wong. Some properties of query languages for bags. In *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 97–114. Springer-Verlag, January 1994.

[13] N. Mattos and L. G. DeMichiel. Recent Design Trade-offs in SQL3. *SIGMOD Record*, 23(4):84–89, December 1994.

[14] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli, a polymorphic language with static type inference. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, June 1989.

[15] J. Paredaens and D. Van Gucht. Converting nested relational algebra expressions into flat algebra expressions. *ACM Transaction on Database Systems*, 17(1):65–93, March 1992.

[16] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proceedings of 16th Symposium on Principles of Programming Languages*, pages 77–88, 1989.

[17] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[18] D. Suciu and J. Paredaens. Any algorithm in the complex object algebra needs exponential space to compute transitive closure. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 201–209, Minneapolis, Minnesota, May 1994.

[19] D. Suciu and L. Wong. On two forms of structural recursion. In *Proceedings of 5th International Conference on Database Theory*, pages 111–124, Prague, January 1995.

[20] S. J. Thomas and P. C. Fischer. Nested relational structures. In *Advances in Computing Research: The Theory of Databases*, pages 269–307, London, England, 1986. JAI Press.

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

[22] P. Wadler. List comprehensions. In *The Implementation of Functional Languages*. Prentice Hall, 1987.

[23] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[24] M. Wand. Complete type inference for simple objects. In *Proceedings of 2nd IEEE Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, June 1987.

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

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