MATISSE: OPL Redesigned as an Object Query Language*

Hassan Aït-Kaci

ILOG, S.A.
B.P. 85 - 9, rue de Verdun
94253 Gentilly, France

hak@ilog.fr

February 18, 2002

Abstract

In this document, I undertake the design of the declarative part of an OPL-like language (here, provisionally called "MATISSE") using formal concepts from the state of the art in object-oriented database query languages. I start with an analysis of the constructs (data structures and operations) used by OPL for defining a model, arguing that the declarative part of OPL may be viewed essentially as a query language over complex data structures, including collections (such as lists, arrays, sets, and bags). In order to explicate the precise meaning and optimized implementation of expressions involving these constructs (such as arbitrarily nested aggregates, joins, and user-defined functions), I proceed to review prominent proposals from recent research in object query language design purporting to embody formally and efficiently queries over objects and collections thereof, and discuss their merits and shortcomings for the specific needs of a language such as MATISSE. I then adapt some of these ideas to propose a core algebra and calculus with the goal of capturing all the declarative expressivity of OPL without any of its current limitations nor quirks of uniformity, trying to provide the most general, yet efficient, language conveniences.

1 Introduction

Programs expressed in the OPL language [10] typically consist of two separate parts: a model definition and an optional search specification. The model definition is purely declarative and amounts to declaring data structures and setting up a model consisting of set of algebraic constraints with, possibly, an objective "function" to optimize. The search part is not declarative and consists of explicit indications on how to proceed with searching for solutions of a defined model. We are only interested here in the purely declarative part of OPL—the model definition. Upon scrutiny, one realizes that defining a model amounts to declaring data structures and specifying a model as a syntactic object expression constructed out of these structures using data manipulation operations. Thus, the main idea of this work is based on the realization that an OPL model definition can in fact all be formulated as an expression of an object query language [11].

The motivation for redesigning the declarative part of OPL at all stems from the fact that it suffers from several undue shortcomings [15]. While several of these are cosmetic in nature, some

*This is an incomplete draft reporting work in progress.

are the result of not having recognized that the operations and data types for the representation and manipulation of data it needs have been well studied in the area of object query languages. As a result, OPL provides useful conveniences that feel very appealing to the user but unfortunately work only for very limited cases, falling short of supporting these constructs in their full generality. This state of affairs ends up frustrating the user who is led to expect much more flexibility than is actually supported, not understanding what caused a *bona fide* expression to be flagged as disallowed when other similar ones are allowed. My proposal is to attempt redesigning OPL to support in their full generality all the data definition and manipulation conveniences it offers, taking inspiration in recent work in object query language design.

This document is organized as follows. Section 2 briefly recounts what one needs to know about query languages. Section 3 examines OPL's constructs for defining models and argues that it all falls within the object query language paradigm. In Section 4, I present recent work proposing formal algebras and calculi for expressing the precise semantics and efficient implementations of object query languages. In Section 5, I discuss the specific *desiderata* for MATISSE in the light of the reviewed formalisms. In Section 6, I proceed to adapt these ideas to our needs to arrive at an algebra and a calculus sufficient to express precisely OPL constructs in their full generality, to be used as the formal basis for a kernel language into which to "desugar" a more convenient surface syntax.

2 A brief history of query languages

The relational model for databases proposed by Codd [5, 6, 7] owed its resounding success to using a simple, yet powerful, mathematical formalism to capture the representation and manipulation of data. It made a clear distinction between the formal semantics of its constructs and operations (the *Relational Algebra*) and the effective computation of query expressions taking their meaning in the Relational Algebra (the *Relational Calculus*). In its essence, the only mathematical formalism used by Relational Algebra is elementary Set Theory. This enabled expressing simply and elegantly database queries whose semantic correctness is straightforward to establish due to the pure declarativity of the Relational Algebra. It also enabled using the formal algebraic properties of its operations (such as commutativity, associativity, *etc.*) to optimize implementation of the Relation Calculus while provably preserving a query's formal meaning. Thus, the Relational Calculus provides the kernel language in which to express the various constructs of a practical query language based on relations, which are ascribed a precise formal semantics in the Relational Algebra. Such a practical query language has been proposed, and well accepted, as a standard known as the Structured Query Language (SQL).[1]

Object-oriented databases have evolved out of the relational model by allowing more flexibility in the form taken by the data and incorporating the object-oriented notions of type encapsulation and inheritance. The added flexibility consists essentially in allowing arbitrary nesting of record object structures (*i.e.*, tuples) and collections (*i.e.*, sets, bags, lists, *etc.*). In other words, whereas the relational model deals exclusively with relations over tuples of primitive types (*i.e.*, sets of flat tuples—so-called "relational normal forms"), the object model imposes no such restriction. The other added conveniences—(polymorphic) typing, inheritance, and class/method encapsulation—have been introduced for the same obvious reasons as for general-purpose programming languages (*i.e.*, data abstraction, reusability, modularity, *etc.*). As it turns out, the mere task of dealing with non-normal form objects alone in the same simple formal fashion as was done in the relational model (*i.e.*, providing an algebra and a calculus that can be optimized thanks to the algebra's equational theory) has been a hard task to tackle. Only recently have some elegant,

[1] See, *e.g.*, [12] for an excellent, easily accessible tutorial.

yet powerful, formalisms emerged that begin to meet the challenge [1, 16, 17]. In other words, these formalisms are good candidates to stand with respect to OQL, the ODMG Object Query Language standard [2], as the relational algebra and calculus stand with respect to SQL.

While the work done in the past twenty or so years in programming language design theory has made object-orientation and typing an orthogonal issue to the underlying computational model, one does need an underlying calculus to start with (*e.g.*, in order to have C++, one needs to have C first). In other words, given a calculus, be it functional, logical, or imperative, it is not a major task to recast it as an object-oriented calculus by adapting its syntax and typing rules to deal with encapsulation and inheritance, while keeping its underlying computational model. My intention, thus, is to start with a "flat" typed object algebra and calculus rich enough to express and optimize the construction of a constraint model from a high-level definition of the kind supported by OPL, and delay incorporating true object-orientation until such a basic formalism is at hand. Indeed, providing full object-orientation for MATISSE is only a second phase which should unfold relatively straightforwardly once its basic formalism has been designed.

3 OPL as an object query language

Let us consider an OPL program defining a transportation model. A certain set of *products* are manufactured and consumed in a set of *cities*. Data describing the *supply* is given specifying how much of each product is manufactured at each city, how much of each product is consumed at each city, and how much it costs per unit of product to be shipped from each city to each city. We want to establish how much of each good to transport between cities in such a way as to meet the demand within the supply and minimize the total cost.

The complete listing of an OPL program defining a model for a transportation problem is given in Appendix Section B. I will refer to the line numbers of that listing in the following discussion.

4 A review of object query language formalisms

In this section, I review prominent formalisms, focusing mainly on two recent proposals using the notion of comprehension syntax due to Peter Buneman *et al.* [1] with extensions to support arrays—a data structure of prime importance for an language with mathematical programming capabilities such as OPL. The first is due to Leonidas Fegaras and David Maier [9] and uses the notion of *Monoid Comprehensions*; the second is due to Leonid Libkin, Rona Machlin, and Limsoon Wong [14] and proposes a functional calculus of array objects. These two formalisms are very similar, differing mostly in their treatment of arrays (as collections in [9], and as functions in [14]). My intention is to try and merge ideas from both for our purpose.

4.1 Monoid homomorphisms and comprehensions

The formalism presented here is based on [9] and assumes familiarity with the notions and notations summarized in Appendix Section A.2. I will use the programming view of monoids exposed there using the specific notation of monoid attributes, in particular for sets, bags, and lists. I will also assume basic familiarity with naive λ-calculus and associated typing as presented in Appendix Section A.3.

4.1.1 Monoid homomorphisms

Because many operations and data structures are monoids, it is interesting to use the associated concepts as the computational building block of an essential calculus. In particular, iteration over collection types can be elegantly formulated as computing a *monoid homomorphism*. This notion coincides with the usual mathematical notion of homomorphism, albeit given here from an operational standpoint and biased toward collection monoids. Basically, a monoid homomorphism $\mathfrak{hom}_{\oplus}^{\odot}$ maps a function f from a collection monoid $\oplus$ to *any* monoid $\odot$ by collecting all the f-images of elements of a $\oplus$-collection using the $\odot$ operation. For example, the expression $\mathfrak{hom}_{++}^{\cup}[\lambda x.x + 1]$ applied to the list $[1, 2, 1, 3, 2]$ returns the set $\{2, 3, 4\}$. In other words, the monoid homomorphism $\mathfrak{hom}_{++}^{\cup}$ of a function f applied to a list L corresponds to the following *loop* computation:

$$
\begin{array}{l}
\texttt{result} \leftarrow \{\};\\
\textbf{foreach}\ x\ \textbf{in}\ L\ \textbf{do}\ \texttt{result} \leftarrow \texttt{result} \cup f(x);\\
\textbf{return}\ \texttt{result};
\end{array}
$$

This is formulated formally as follows:

DEFINITION 4.1 (MONOID HOMOMORPHISM) *A* Monoid Homomorphism $\mathfrak{hom}_{\oplus}^{\odot}$ *defines a mapping from a* collection *homomorphism* $\oplus$ *to* any *monoid such that* $\Theta_{\oplus} \subseteq \Theta_{\odot}$ *by:*

$$
\begin{array}{lcl}
\mathfrak{hom}_{\oplus}^{\odot}[f](\mathfrak{z}_{\oplus}) & \stackrel{def}{=} & \mathfrak{z}_{\odot}\\
\mathfrak{hom}_{\oplus}^{\odot}[f](\mathfrak{u}_{\oplus}(x)) & \stackrel{def}{=} & f(x)\\
\mathfrak{hom}_{\oplus}^{\odot}[f](x \oplus y) & \stackrel{def}{=} & \mathfrak{hom}_{\oplus}^{\odot}[f](x) \odot \mathfrak{hom}_{\oplus}^{\odot}[f](y)
\end{array}
$$

for any function $f : \alpha \to \mathfrak{T}_{\odot}$*,* $x : \alpha$*, and* $y : \alpha$*, where* $\mathfrak{T}_{\oplus} = \mathfrak{C}_{\oplus}(\alpha)$.

Again, computationally, this amounts to executing the following iteration:

$$
\begin{array}{l}
\texttt{result} \leftarrow \mathfrak{z}_{\odot};\\
\textbf{foreach}\ x_i\ \textbf{in}\ \mathfrak{u}_{\oplus}(x_1) \oplus \cdots \oplus \mathfrak{u}_{\oplus}(x_n)\ \textbf{do}\ \texttt{result} \leftarrow \texttt{result} \odot f(x_i);\\
\textbf{return}\ \texttt{result};
\end{array}
\tag{1}
$$

The reader may be puzzled by the condition $\Theta_{\oplus} \subseteq \Theta_{\odot}$ in Definition 4.1. It means that a monoid homomorphisms may only be defined from a collection monoid to a monoid that has at least the same equational theory. In other words, one can only go from an empty theory monoid, to either a $\{C\}$-monoid or an $\{I\}$-monoid, or yet to a $\{C, I\}$-monoid. This requirement is due to an algebraic technicality, and relaxing it would enable a monoid homomorphism to be ill-defined. To see this, consider going from, say, a commutative-idempotent monoid to one that is commutative but not idempotent. Let us take, for example, $\mathfrak{hom}_{\cup}^{+}$. Then, this entails:

$$
\begin{aligned}
1 &= \mathfrak{hom}_{\cup}^{+}[\lambda x.1](\{a\})\\
&= \mathfrak{hom}_{\cup}^{+}[\lambda x.1](\{a\} \cup \{a\})\\
&= \mathfrak{hom}_{\cup}^{+}[\lambda x.1](\{a\}) \ + \ \mathfrak{hom}_{\cup}^{+}[\lambda x.1](\{a\})\\
&= 1 + 1\\
&= 2.
\end{aligned}
$$

The reader may have noticed that this restriction has the unfortunate consequence of disallowing potentially useful computations, notable examples being computing the cardinality of a set, or converting a set into a list. However, this drawback can be easily overcome with a suitable modification of the third clause in Definition 4.1, and other expressions based on it, ensuring that anomalous cases such as the above are dealt with by appropriate tests.

Also of importance for the consistency of Definition 4.1 is the fact that a non-idempotent monoid must be anti-idempotent, and a non-commutative monoid must be anti-commutative. Indeed, if $\oplus$ is non-idempotent as well as non-anti-idempotent (say, $x_0 \oplus x_0 = x_0$ for some x_0), then this entails:

$$\begin{aligned}\mathfrak{hom}_{\oplus}^{\odot}[f](x_0) &= \mathfrak{hom}_{\oplus}^{\odot}[f](x_0 \oplus x_0) \\ &= \mathfrak{hom}_{\oplus}^{\odot}[f](x_0) \ \odot \ \mathfrak{hom}_{\oplus}^{\odot}[f](x_0)\end{aligned}$$

which is not necessarily true for non-idempotent $\odot$. A similar argument may be given for commutativity. This consistency condition is in fact not restrictive operationally as it is always verified (*e.g.*, a list will not allow partial commutation of any of its element).

Here are a few familar functions expressed with well-defined monoid homomorphisms:

$$\begin{aligned}\mathtt{length}(l) \quad &= \mathfrak{hom}_{++}^{+}[\lambda x.1](l) \\ e \in s \quad &= \mathfrak{hom}_{\cup}^{\vee}[\lambda x.x = e](s) \\ s \times t \quad &= \mathfrak{hom}_{\cup}^{\cup}[\lambda x.\mathfrak{hom}_{\cup}^{\cup}[\lambda y.\{\langle x, y\rangle\}](t)](s) \\ \mathtt{map}(f, s) \quad &= \mathfrak{hom}_{\cup}^{\cup}[\lambda x.\{f(x)\}](s) \\ \mathtt{filter}(p, s) &= \mathfrak{hom}_{\cup}^{\cup}[\lambda x.\mathfrak{if}\ p(x)\ \mathfrak{then}\ \{x\}\ \mathfrak{else}\ \{\}](s).\end{aligned}$$

4.1.2 Monoid comprehensions

The concept of monoid homomorphism is useful for expressing a formal semantics of iteration over collections. However, it is not very convenient as a programming construct. A natural notation for such a construct that is both conspicuous and can be expressed in terms of monoid homomorphisms is a *monoid comprehension*. This notion generalizes the familiar notation used for writing a set in comprehension (as opposed to writing it in extension) using a pattern and a formula describing its elements (as oppposed to listing all its elements). For example, the set comprehension $\{\langle x, x^2\rangle \mid x \in \mathbb{N}, \exists n.x = 2n\}$ describes the set of pairs $\langle x, x^2\rangle$ (the *pattern*), verifying the formula $x \in \mathbb{N}, \exists n.x = 2n$ (the *qualifier*).

This notation can be extended to any (primitive or collection) monoid $\oplus$. The syntax of a monoid comprehension is an expression of the form $\oplus\{e \parallel Q\}$ where e is an expression called the *head* of the comprehension, and Q is called its qualifier and is a sequence $q_1, \ldots, q_n$, $n \geq 0$, where each q_i is either

- a *generator* of the form $x \leftarrow e$, where x is a variable and e is an expression; or,
- a *filter* ϕ which is a boolean condition.

In a monoid comprehension expression $\oplus\{e \parallel Q\}$, the monoid operation $\oplus$ is called the *accumulator*.

As for semantics, the meaning of a monoid comprehension is defined in terms of monoid homomorphisms.

DEFINITION 4.2 (MONOID COMPREHENSION) *The meaning of a monoid comprehension over a monoid $\oplus$ is defined inductively as follows:*

$$\oplus\{e \parallel \} \stackrel{\mathit{def}}{=} \begin{cases} \mathfrak{u}_{\oplus}(e) & \textit{if } \oplus \textit{ is a collection monoid} \\ e & \textit{if } \oplus \textit{ is a primitive monoid} \end{cases}$$

$$\oplus\{e \parallel x \leftarrow e', Q\} \stackrel{\mathit{def}}{=} \mathfrak{hom}^{\oplus}_{\odot}[\lambda x. \oplus \{e \parallel Q\}](e')$$

$$\oplus\{e \parallel c, Q\} \stackrel{\mathit{def}}{=} \mathfrak{if}\ c\ \mathfrak{then}\ \oplus \{e \parallel Q\}\ \mathfrak{else}\ \mathfrak{z}_{\oplus}$$

such that $e : \mathfrak{T}_{\oplus}$, $e' : \mathfrak{T}_{\odot}$, and $\odot$ is a collection monoid.

Note that although the input monoid $\oplus$ is explicit, each generator $x \leftarrow e'$ in the qualifier has an implicit collection monoid $\odot$ whose characteristics can be inferred with polymorphic typing rules.

Although Definition 4.2 can be effectively computed using nested loops (*i.e.*, using the iteration semantics (1)), such would be in general rather inefficient. Rather, an optimized implementation can be achieved by various syntactic transformation expressed as rewrite rules. Thus, the principal benefit of using monoid comprehensions is to formulate efficient optimizations on a simple and uniform general syntax of expressions irrespective of specific monoids.

Note that relational *joins* are immediately expressible as monoid comprehensions. Indeed, the join of two sets S and T using a function f and a predicate p is simply:

$$S \bowtie^{f}_{p} T \stackrel{\mathrm{def}}{=} \cup\{f(x,y) \parallel x \leftarrow S, y \leftarrow T, p(x,y)\}. \tag{2}$$

Typically, a relational join will take f to be a record constructor. For example, if we write a record whose fields $\mathtt{l}_i$ have values e_i for $i = 1, \ldots, n$, as $\langle \mathtt{l}_1 = e_1, , \ldots, \mathtt{l}_n = e_n \rangle$, then a standard relational join is obtained with, say, $f(x,y) = \langle \mathtt{name} = y.\mathtt{name}, \mathtt{age} = 2 * x.\mathtt{age} \rangle$, and $p(x,y)$ may be any condition such as $x.\mathtt{name} = y.\mathtt{name}, x.\mathtt{age} \geq 18$.

Clearly, monoid comprehensions can immediately express queries using all usual relational operators (and, indeed, object queries as well) and most usual functions. For example,

$$\begin{array}{lcllcl}
\exists x \in s.e & \stackrel{\mathrm{def}}{=} & \vee\{e \parallel x \leftarrow s\} & \mathtt{length}(s) & \stackrel{\mathrm{def}}{=} & +\{1 \parallel x \leftarrow s\} \\
\forall x \in s.e & \stackrel{\mathrm{def}}{=} & \wedge\{e \parallel x \leftarrow s\} & \mathtt{sum}(s) & \stackrel{\mathrm{def}}{=} & +\{x \parallel x \leftarrow s\} \\
x \in s & \stackrel{\mathrm{def}}{=} & \vee\{x = y \parallel y \leftarrow s\} & \mathtt{max}(s) & \stackrel{\mathrm{def}}{=} & \max\{x \parallel x \leftarrow s\} \\
s \cap t & \stackrel{\mathrm{def}}{=} & \cup\{x \parallel x \leftarrow s, x \in t\} & \mathtt{filter}(p,s) & \stackrel{\mathrm{def}}{=} & \cup\{x \parallel x \leftarrow s, p(x)\} \\
\mathtt{count}(a,s) & \stackrel{\mathrm{def}}{=} & +\{1 \parallel x \leftarrow s, x = a\} & \mathtt{flatten}(s) & \stackrel{\mathrm{def}}{=} & \cup\{x \parallel t \leftarrow s, x \leftarrow t\}
\end{array}$$

Note that some of these functions will work only on appropriate types of their arguments. For example, the type of the argument of `sum` must be a non-idempotent monoid, and so must the type of the second argument of `count`. Thus, `sum` will add up the elements of a bag or a list, and `count` will tally the number of occurrences of an element in a bag or a list. Applying either `sum` or `count` to a set will be caught as a type error.

4.1.3 The monoid comprehension calculus

We are now in a position to propose a programming calculus using monoid comprehensions. Figure 1 defines an abstract grammar for an expression e of the *Monoid Comprehension Calculus* and amounts to adding comprehensions to an extended Typed Polymorphic λ-Calculus. Figure 2 gives the typing rules for this calculus.

$$
\begin{array}{rcll}
e & ::= & \perp & \text{null value} \\
 & | & c & \text{constant} \\
 & | & x & \text{symbol} \\
 & | & \lambda x.e & \text{abstraction} \\
 & | & e_1\ e_2 & \text{application} \\
 & | & \langle \mathtt{l}_1 = e_1, \cdots, \mathtt{l}_n = e_n \rangle & \text{record} \\
 & | & e.\mathtt{l} & \text{projection} \\
 & | & \mathfrak{if}\ e_1\ \mathfrak{then}\ e_2\ \mathfrak{else}\ e_3 & \text{conditional} \\
 & | & \mathfrak{z}_{\oplus} & \text{monoid identity} \\
 & | & \mathfrak{u}_{\oplus}(e) & \text{monoid unit injection} \\
 & | & e_1 \oplus e_2 & \text{monoid operation} \\
 & | & \oplus\{e \parallel Q\} & \text{monoid comprehension}
\end{array}
$$

Figure 1: The monoid comprehension calculus

4.2 Multidimensional arrays as functions

5 Discussion

5.1 What types for MATISSE?

Before we delve into redesigning a new OPL, we need to ponder its type particularities—indeed, peculiarities. The current OPL departs from conventional programming langauges when it comes to types in several respects. Figure 3 shows the types supported by OPL.

5.1.1 Primitive types

5.1.2 Object types

5.1.3 Collections types

5.2 Array types

5.2.1 First-class indexing

HAK NOTE: *Discuss the ZPL-region approach [3] for use in* MATISSE*, and their sparse version [4].*

$$\frac{}{\Gamma \vdash \bot : \alpha} \quad \text{for any } \Gamma$$

$$\frac{}{\Gamma \vdash c : \tau} \quad \text{for any } \Gamma\text{, if } \mathbf{type}(a) = \tau$$

$$\frac{}{\Gamma \vdash x : \tau} \quad \text{if } \Gamma(x) = \tau$$

$$\frac{\Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2,\ \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1, \cdots, \Gamma \vdash e_k : \tau_k}{\Gamma \vdash \langle \mathtt{l}_1 = e_1, \cdots, \mathtt{l}_k = e_k \rangle : \langle \mathtt{l}_1 : \tau_1, \cdots, \mathtt{l}_k : \tau_k \rangle}$$

$$\frac{\Gamma \vdash e : \langle \mathtt{l}_1 : \tau_1, \cdots, \mathtt{l}_k : \tau_k \rangle}{\Gamma \vdash e.\mathtt{l} : \tau} \quad \text{if } \mathtt{l} \in \{\mathtt{l}_1, \ldots, \mathtt{l}_k\}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{bool},\ \Gamma \vdash e_2 : \tau,\ \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathfrak{if}\ e_1\ \mathfrak{then}\ e_2\ \mathfrak{else}\ e_3 : \tau}$$

$$\frac{}{\Gamma \vdash \mathfrak{z}_\oplus : \mathfrak{T}_\oplus} \quad \text{for any } \Gamma$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathfrak{u}_\oplus(e) : \mathfrak{C}_\oplus(\tau)} \quad \text{if } \oplus \text{ is a collection monoid}$$

$$\frac{\Gamma \vdash e_1 : \mathfrak{T}_\oplus,\ \Gamma \vdash e_2 : \mathfrak{T}_\oplus}{\Gamma \vdash e_1 \oplus e_2 : \mathfrak{T}_\oplus} \quad \text{if } \oplus \text{ is a primitive monoid}$$

$$\frac{\Gamma \vdash e_1 : \mathfrak{C}_\oplus(\tau),\ \Gamma \vdash e_2 : \mathfrak{C}_\oplus(\tau)}{\Gamma \vdash e_1 \oplus e_2 : \mathfrak{C}_\oplus(\tau)} \quad \text{if } \oplus \text{ is a collection monoid}$$

$$\frac{\Gamma \vdash e : \mathfrak{T}_\oplus}{\Gamma \vdash \oplus\{e \parallel\ \} : \mathfrak{T}_\oplus} \quad \text{if } \oplus \text{ is a primitive monoid}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \oplus\{e \parallel\ \} : \mathfrak{C}_\oplus(\tau)} \quad \text{if } \oplus \text{ is a collection monoid}$$

$$\frac{\Gamma \vdash e_2 : \mathfrak{C}_\odot(\tau_2),\ \Gamma[x : \tau_2] \vdash \oplus\{e_1 \parallel Q\} : \tau_1}{\Gamma \vdash \oplus\{e_1 \parallel x \leftarrow e_2, Q\} : \tau_1} \quad \text{if } \Theta_\odot \subseteq \Theta_\oplus$$

$$\frac{\Gamma \vdash e_2 : \mathtt{bool},\ \Gamma \vdash \oplus\{e_1 \parallel Q\} : \tau}{\Gamma \vdash \oplus\{e_1 \parallel e_2, Q\} : \tau}$$

Figure 2: Typing rules for the Monoid Comprehension Calculus

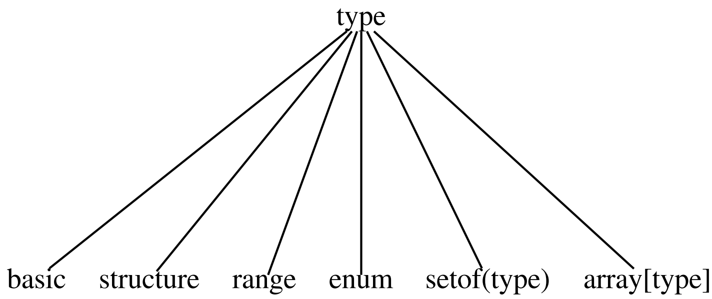

Figure 3: OPL types

5.2.2 Indexing types

5.3 Functional abstraction

6 Designing MATISSE

6.1 Algebra

6.2 Semantics

6.3 Implementation

Appendix

In this part of the document, I summarize well-known notions that may be needed by some readers lacking the appropriate background. This is in order to spare those who already know this material and unclutter the main body of the text. The latter reason is also why Section B contains the complete listing of the transportation model used in the discussion of Section 3.

A Technical Background

A.1 The Relational Model

A.2 Monoids

In this section, all notions and notations relating to monoids as they are used in this paper are recalled and justified.

Mathematically, a monoid is a non-empty set equipped with an associative internal binary operation and an identity element for this operation. Formally, let S be a set, $\star$ a function from $S \times S$ to S, and $\epsilon \in S$; then, $\langle S, \star, \epsilon \rangle$ is a monoid iff, for any x, y, z in S:

$$x \star (y \star z) = (x \star y) \star z \tag{3}$$

and

$$x \star \epsilon = \epsilon \star x = \epsilon. \tag{4}$$

Most familiar mathematical binary operations define monoids. For example, taking the set of natural numbers $\mathbb{N}$, and the set of boolean values $\mathbb{B} = \{\mathfrak{true}, \mathfrak{false}\}$, the following are monoids:

- $\langle \mathbb{N}, +, 0 \rangle$,

- $\langle \mathbb{N}, *, 1 \rangle$,
- $\langle \mathbb{N}, \max, 0 \rangle$,
- $\langle \mathbb{B}, \vee, \mathfrak{false} \rangle$,
- $\langle \mathbb{B}, \wedge, \mathfrak{true} \rangle$.

The operations of these monoids are so familiar that they need not be explicated. For us, they have a "built -in" semantics that allows us to compute with them since primary school. Indeed, we shall refer to such readily interpreted monoids as *primitive monoids*.[2]

Note that the definition of a monoid does not preclude additional algebraic structure. Such structure may be specified by other equations augmenting the basic monoid equational theory given by the conjunction of equations (3) and (4). For example, all five monoids listed above are *commutative*; namely, they also obey equation (5):

$$x \star y = y \star x \tag{5}$$

for any x, y. In addition, the three last ones (*i.e.*, $\max$, $\vee$, and $\wedge$) are also *idempotent*; namely, they also obey equation (6):

$$x \star x = x \tag{6}$$

for any x.

Not all monoids are primitive monoids. That is, one may define a monoid purely syntactically whose operation only builds a syntactic structure rather than being interpreted using some semantic computation. For example, linear lists have such a structure: the operation is list concatenation and builds a list out of two lists; its identity element is the empty list. A syntactic monoid may also have additional algebraic structure. For example, the monoid of bags is also defined as a commutative syntactic monoid with the disjunct union operation and the empty bag as identity. Or, the monoid of sets is a commutative and idempotent syntactic monoid with the union operation and the empty set as identity.

Because they are not interpreted, syntactic monoids pose a problem as far as representation of its elements is concerned. To illustrate this, let us consider an empty-theory algebraic structure; that is, one without any equations—not even associativity nor identity. Let us take such a structure with one binary operation $\star$ on, say, the natural numbers $\mathbb{N}$. Saying that $\star$ is a "syntactic" operation means that it constructs a syntactic term (*i.e.*, an expression tree) by composing two other syntactic terms. We thus can define the set $T_\star$ of $\star$-terms on some base set, say the natural numbers, inductively as the limit $\cup_{n \geq 0} T_n$ where,

$$T_n \stackrel{\text{def}}{=} \begin{cases} \mathbb{N} & \text{if } n = 0 \\ \\ \{t_1 \star t_2 \mid t_i \in T_{n-1}, i = 1, 2\} & \text{if } n > 0 \end{cases} \tag{7}$$

Clearly, the set $T_\star$ is well defined and so is the $\star$ operation over it. Indeed, $\star$ is a *bona fide* function from $T_\star \times T_\star$ to $T_\star$ mapping two terms t_1 and t_2 in $T_\star$ into a unique term in $T_\star$—namely, $t_1 \star t_2$. This is why $T_\star$ is called the *syntactic* algebra.[3]

[2]We call these monoids "primitive" following the presentation of Fegaras and Maier [9] as it adheres to a more operational (as opposed to mathematical) approach more suitable to computer-scientists. Mathematically, however, these should be called "semantic" monoids since they are interpreted by the computation semantics of their operations.

[3]For a fixed set of base elements and operations (which constitute what is formally called a *signature*), the syntactic algebra is unique (up to isomorphism). This algebra is also called the *free*, or the *initial*, algebra for its signature.

Let us now assume that the $\star$ operation is associative—*i.e.*, that $\star$-terms verify Equation (3). Note that this equation defines a (syntactic) *congruence* on $T_\star$ which identifies terms such as, say, $1 \star (2 \star 3)$ and $(1 \star 2) \star 3$. In fact, for such an associative $\star$ operation, the set $T_\star$ defined in Equation (7) is not the appropriate domain. Rather, the right domain is the quotient set whose elements are (syntactic) congruence classes modulo associativity of $\star$. Therefore, this creates an ambiguity of representation of the syntactic structures.[4]

Similarly, more algebraic structure defined by larger equational theories induces coarser quotients of the empty-theory algebra by putting together in common congruence classes all the syntactic expressions that can be identified modulo the theory's equations. The more equations, the more ambiguous the syntactic structures of expressions. Mathematically, this poses no problem as one can always abstract away from individuals to congruence classes. However, operationally one must resort to some concrete artifact to obtain a unique representation for all members of the same congruence class. One way is to devise a *canonical* representation into which to transform all terms. For example, an associative operation could systematically "move" nested subtrees from its left argument to its right argument—in effect using Equation (3) as a one-way rewrite rule. However, while this is possible for some equational theories, it is not so in general—*e.g.*, take commutativity.[5]

From a programming standpoint (which is ours), we can abstract away from the ambiguity of canonical representations of syntactic monoid terms using a flat notation. For example, in LISP and Prolog, a list is seen as the (flat) sequence of its constituents. Typically, a programmer writes $[1, 2, 1]$ to represent the list whose elements are 1, 2 and 1 in this order, and does not care (nor need s/he be aware) of its concrete representation. A set—*i.e.*, a commutative idempotent syntactic monoid—is usually denoted by the usual mathematical notation $\{1, 2\}$, implicitly relying on disallowing duplicate elements, not minding the order in which the elements appear. A bag, or multiset—*i.e.*, a commutative but non-idempotent syntactic monoid—uses a similar notation, allowing duplicate elements but paying no heed to the order in wich they appear; *i.e.*, $\{\!\{1, 2, 1\}\!\}$ is the bag containing 1 twice, and 2 once.

Syntactic monoids are quite useful for programming as they provide adquate data structures to represent collections of objects of a given type. Thus, we refer to them as *collection monoids*. Now, a definition such as Equation (7) for a syntactic monoid, although sound mathematically, is not quite adequate for programming purposes. This is because it defines the $\star$ operations on two distinct *types* of elements; namely, the base elements (here natural numbers) and constructed elements. In programming, it is desirable that operations be given a crisp type. A way to achieve this is by systematically "wrapping" each base element x into a term such as $x \star \epsilon$. This "wrapping" is achieve by associating to the monoid a function $\mathfrak{u}_\star$ from the base set into the monoid domain called its *unit injection*. For example, if $+\!\!+$ is the list monoid operation for concatenating two lists, $\mathfrak{u}_{+\!+}(x) = [x]$ and one may view the list $[a, b, c]$ as $[a] +\!\!+ [b] +\!\!+ [c]$. Similarly, the set $\{a, b, c\}$ is viewed as $\{a\} \cup \{b\} \cup \{c\}$, and the bag $\{\!\{a, b, c\}\!\}$ as $\{\!\{a\}\!\} \uplus \{\!\{b\}\!\} \uplus \{\!\{c\}\!\}$. Clearly, this bases the constructions on an isomorphic view of the base set rather than the base set itself, while using a uniform type for the monoid operator. Also, because the type of the base elements is irrelevant for the construction other than imposing the constraint that all such elements be of the same type, we present a collection monoid as a *polymorphic* data type. This justifies the formal view of monoids we give next using the programming notion of type.

[4]Note that this ambiguity never arises for semantic algebras whose operations are interpreted into a unique result.

[5]Such are important considerations in the field of *term rewriting* [8], where the problem of finding canonical term representations for equational theories was originally addressed by Donald Knuth and Peter Bendix in a seminal paper proposing a general effective method—the so-called Knuth-Bendix Completion Algorithm [13]. The problem, incidentally, is only semi-decidable. In other words, the Knuth-Bendix algorithm may diverge, although several interesting variations have been proposed for a wide extent of practical uses (see [8] for a good introduction and bibliography).

$\oplus$	$\mathfrak{T}_\oplus$	$\mathfrak{z}_\oplus$	$\Theta_\oplus$
$+$	`int`	0	$\{C\}$
$*$	`int`	1	$\{C\}$
$\max$	`int`	0	$\{C, I\}$
$\vee$	`bool`	$\mathfrak{false}$	$\{C, I\}$
$\wedge$	`bool`	$\mathfrak{true}$	$\{C, I\}$

Primitive monoids

$\oplus$	$\mathfrak{C}_\oplus$	$\mathfrak{T}_\oplus$	$\mathfrak{z}_\oplus$	$\mathfrak{u}_\oplus(x)$	$\Theta_\oplus$
$\cup$	`set`	`set`(α)	$\{\}$	$\{x\}$	$\{C, I\}$
$\uplus$	`bag`	`bag`(α)	$\{\!\{\}\!\}$	$\{\!\{x\}\!\}$	$\{C\}$
$+\!\!+$	`list`	`list`(α)	$[]$	$[x]$	$\emptyset$

Collection monoids

Table 1: Attributes of a few common monoids

Because it is characterized by its operation $\oplus$, a monoid is often simply referred to as $\oplus$. Thus, a monoid operation is used as a subscript to denote its characteristic attributes. Namely, for a monoid $\oplus$,

- $\mathfrak{T}_\oplus$ is its type (*i.e.*, $\oplus : \mathfrak{T}_\oplus \times \mathfrak{T}_\oplus \to \mathfrak{T}_\oplus$),
- $\mathfrak{z}_\oplus : \mathfrak{T}_\oplus$ is its identity element,
- $\Theta_\oplus$ is its equational theory (*i.e.*, a subset of the set $\{C, I\}$, where C stands for "commutative" and I for "idempotent");

and, if it is a collection monoid,

- $\mathfrak{C}_\oplus$ is its type constructor (*i.e.*, $\mathfrak{T}_\oplus = \mathfrak{C}_\oplus(\alpha)$),
- $\mathfrak{u}_\oplus : \alpha \to \mathfrak{C}_\oplus(\alpha)$ is its unit injection.

Table 1 summarizes the monoid attributes of a few usual monoids.

A.3 The Typed Polymorphic λ-Calculus

We assume a set $\mathbf{C}$ of pregiven constants ususally denoted by $a, b \ldots$, and a countably infinite set of variable symbols $\mathbf{V}$ usually denoted by $x, y, \ldots$. The syntax of a term t of the λ-Calculus is given by the following grammar:

$$\begin{array}{rclll} t & ::= & a & (a \in \mathbf{C}) & \text{constant} \\ & | & x & (x \in \mathbf{V}) & \text{symbol} \\ & | & \lambda x.t & (x \in \mathbf{V}) & \text{abstraction} \\ & | & t\ t & & \text{application} \end{array} \tag{8}$$

We shall call $\mathbf{T}$ the set of terms t defined by this grammar. These terms are also called *raw* terms.

An abstraction $\lambda x.t$ defines a *lexical scope* for its *bound variable* x, whose extent is its *body* t. Thus, the notion of free occurrence of a variable in a term is defined as usual, and so is the operation $t_1[t_2/x]$ of substituting a term t_2 for all the free occurrences of a variable x in a term t_1. Thus, a bound variable may be renamed to a new one in its scope without changing the abstraction.

The computation rule defined on λ-terms is the so-called β-reduction:

$$(\lambda x.t_1)\ t_2 \quad \longrightarrow \quad t_1[t_2/x]. \tag{9}$$

We assume a set $\mathcal{B}$ of basic type symbols denoted by $A, B, \ldots$, and a countably infinite set of type variables $\mathcal{V}$ denoted by $\alpha, \beta, \ldots$. The syntax of a type τ of the Typed Polymorphic λ-Calculus is given by the following grammar:

$$\begin{array}{rcllr} \tau & ::= & A & (A \in \mathcal{B}) & \text{basic type} \\ & \mid & \alpha & (\alpha \in \mathcal{V}) & \text{variable type} \\ & \mid & \tau \to \tau & & \text{function type} \end{array} \qquad (10)$$

We shall call $\mathcal{T}$ the set of types τ defined by this grammar. A *monomorphic type* is a type that contains no variable types. Any type containing at least one variable type is called a *polymorphic type*.

The terms of the Typed Polymorphic λ-Calculus are only those raw terms in $\mathbf{T}$ that admit a well-defined type in $\mathcal{T}$. Each constant in $\mathbf{C}$ is assumed to have a unique type in $\mathcal{T}$, and we write $\mathbf{type}(a) = \tau$ to mean that constant a has type τ.

One may assign types to symbols using a *type context* Γ, which is a partial function from V to $\mathcal{T}$. Given a type context Γ, a variable $x \in \mathbf{V}$ and a type $\tau \in \mathcal{T}$, we define:

$$\Gamma[x : \tau](y) \stackrel{\text{def}}{=} \begin{cases} \tau & \text{if } x = y; \\ \\ \Gamma(y) & \text{if } x \neq y. \end{cases} \qquad (11)$$

In other words, $\Gamma[x : \tau]$ coincides with Γ everywhere on $\mathbf{V}$ except at x, where it takes the value τ. The *empty context* is the function noted $\emptyset$ defined nowhere on $\mathbf{V}$.

To find out whether a term t in $\mathbf{T}$ is well-typed, one must exhibit a type assignment that constitutes a suitable typing context for the variable symbols occurring in t, and from which one may ascribe a (unique) type for the whole term. Given a type context Γ, a term t, and a type τ, a *type judgement* is an expression of the form $\Gamma \vdash t : \tau$, which stands to mean that t has been deemed to have type τ when the symbols occurring in it have the types assigned to them by the context Γ. Without a type context Γ, the expression $t : \tau$ is used to mean that $\Gamma \vdash t : \tau$ for some Γ.

Deriving types for terms is done using *typing rules* of the form:

$$\frac{\Gamma_1 \vdash t_1 : \tau_n, \; \cdots \; \Gamma_n \vdash t_n : \tau_n}{\Gamma \vdash t : \tau}$$

which is read as follows: "t has type τ under Γ if t_1 has type τ_1 under $\Gamma_1, \ldots$, and t_n has type τ_n under Γ_n." Note that it is possible that $n = 0$, in which case the rule is written:

$$\frac{}{\Gamma \vdash t : \tau}$$

and it is called an *axiom*.

The typing rules for the Typed Polymorphic λ-Calculus are given in Figure 4. These rules can be readily translated into a Logic Programming language based on Horn-clauses such as Prolog, and used as an effective means to infer the types of expressions based on the Typed Polymorphic λ-Calculus.

The basic syntax of the Typed Polymorphic λ-Calculus may be extended with other operators and convenient data structures as long as typing rules for the new constructs are provided. Typically, one provides at least the set $\mathbb{N}$ of integer constants and $\mathbb{B} = \{\mathfrak{true}, \mathfrak{false}\}$ of boolean constants, along with basic arithmetic and boolean operators, pairing (or tupling), a conditional operator, and a fix-point operator. The usual arithmetic and boolean operators are denoted by constant symbols (*e.g.*, $+, *, -, /, \vee, \wedge$, *etc.*). Let $\mathbf{O}$ be this set.

$$\frac{}{\Gamma \vdash a : \tau} \quad \text{for any } \Gamma, \text{ if } \mathbf{type}(a) = \tau$$

$$\frac{}{\Gamma \vdash x : \tau} \quad \text{if } \Gamma(x) = \tau$$

$$\frac{\Gamma[x : \tau_1] \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2, \; \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2}$$

Figure 4: Typing rules for the Typed Polymorphic λ-Calculus

The computation rules for these operators are based on their usual semantics as one might expect, modulo transforming the usual binary infix notation to a "curryed" application. For example, $t_1 + t_2$ is implicitly taken to be the *application* $(+\ t_1)\ t_2$. Note that this means that all such operators are implicitly "curryed."[6]

For example, we may augment the grammar for the terms given in (8) as follows:

$$\begin{array}{rcllr} t & ::= & a & (a \in \mathbf{C} = \mathbb{N} \cup \mathbb{B} \cup \mathbf{O}) & \text{constant} \\ & | & x & (x \in \mathbf{V}) & \text{symbol} \\ & | & \lambda x.t & (x \in \mathbf{V}) & \text{abstraction} \\ & | & t\ t & & \text{application} \\ & | & \langle t, \cdots, t \rangle & & \text{tupling} \\ & | & t.n & (n \in \mathbb{N}) & \text{projection} \\ & | & \mathfrak{if}\ t\ \mathfrak{then}\ t\ \mathfrak{else}\ t & & \text{conditional} \\ & | & \mathfrak{fix}\ t & & \text{recursion} \end{array} \tag{12}$$

The computation rules for the other new constructs are:

$$\begin{array}{rcl} \langle t_1, \cdots, t_k \rangle.i & \longrightarrow & \begin{cases} t_i & \text{if } 1 \leq i \leq k \\ \text{undefined otherwise} & \end{cases} \\ \mathfrak{if}\ c\ \mathfrak{then}\ t_1\ \mathfrak{else}\ t_2 & \longrightarrow & \begin{cases} t_1 & \text{if } c = \mathfrak{true} \\ t_2 & \text{if } c = \mathfrak{false} \\ \text{undefined otherwise} & \end{cases} \\ \mathfrak{fix}\ t & \longrightarrow & t\ (\mathfrak{fix}\ t) \end{array} \tag{13}$$

To account for the new constructs, the syntax of types is extended accordingly to:

$$\begin{array}{rclll} \tau & ::= & \mathtt{int} \mid \mathtt{bool} & & \text{basic type} \\ & | & \alpha & (\alpha \in \mathcal{V}) & \text{variable type} \\ & | & \langle \tau, \cdots, \tau \rangle & & \text{tuple type} \\ & | & \tau \rightarrow \tau & & \text{function type} \end{array} \tag{14}$$

[6]Recall that a curryed form of an n-ary function f is obtained when f is applied to less arguments than it expects; *i.e.*, $f(t_1, \ldots, t_k)$, for $1 \leq k < n$. In the λ-calculus, this form is simply interpreted as the *abstraction* $\lambda x_1. \ldots \lambda x_{n-k}.f(t_1, \ldots, t_k, x_1, \ldots, x_{n-k})$. In their fully curried form, all n-ary functions can be seen as unary functions; indeed, with this interpretation of curried forms, it is clear that $f(t_1, \ldots, t_n) = (\ldots(f\ t_1) \ldots t_{n-1})\ t_n$.

$$\frac{}{\Gamma \vdash a : \tau} \quad \text{for any } \Gamma \text{, if } \mathbf{type}(a) = \tau$$

$$\frac{}{\Gamma \vdash x : \tau} \quad \text{if } \Gamma(x) = \tau$$

$$\frac{\Gamma[x : \tau_1] \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2,\ \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1,\ \cdots,\ \Gamma \vdash t_k : \tau_k}{\Gamma \vdash \langle t_1, \cdots, t_k \rangle : \langle \tau_1, \cdots, \tau_k \rangle}$$

$$\frac{\Gamma \vdash t : \langle \tau_1, \cdots, \tau_k \rangle}{\Gamma \vdash t.n : \tau} \quad \text{if } 1 \leq n \leq k$$

$$\frac{\Gamma \vdash t_1 : \mathtt{bool},\ \Gamma \vdash t_2 : \tau,\ \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathfrak{if}\ t_1\ \mathfrak{then}\ t_2\ \mathfrak{else}\ t_3 : \tau}$$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash \mathfrak{fix}\ t : \tau}$$

Figure 5: Typing rules for an Extended Typed Polymorphic λ-Calculus

We are given that $\mathbf{type}(n) = \mathtt{int}$ for all $n \in \mathbb{N}$ and that $\mathbf{type}(\mathfrak{true}) = \mathtt{bool}$ and $\mathbf{type}(\mathfrak{false}) = \mathtt{bool}$. The (fully curried) types of the built-in operators are given similarly; *i.e.*, $\mathbf{type}(+) = \mathtt{int} \to (\mathtt{int} \to \mathtt{int})$, $\mathbf{type}(\vee) = \mathtt{bool} \to (\mathtt{bool} \to \mathtt{bool})$, *etc.*, …The typing rules for this extended calculus are given in Figure 5.

B A Transportation Model in OPL

```
enum City ...;
enum Product ...;
float+ limit = ...;

struct TableRoutesType
   { Product p;
     City o;
     City d;
     float+ cost;
   };

{TableRoutesType} TableRoutes = ...;

struct Connection
   { City o;
     City d;
   };

struct Route
```

```
      { Product p;
        Connection e;
      };

   {Route} Routes = { < p,<o,d> > | <p,o,d,c> in TableRoutes };

   {Connection} Connections = { c | <p,c> in Routes };

   struct Supply
      { Product p;
        City o;
      };

   {Supply} Supplies = { <p,c.o> | <p,c> in Routes };

   float+ supply[Supplies] = ...;

   struct Customer
      { Product p;
        City d;
      };

   {Customer} Customers = { <p,c.d> | <p,c> in Routes };

   float+ demand[Customers] = ...;

   float+ cost[Routes];

   initialize
      { forall( <p,o,d,c> in TableRoutes)
           cost[< p,<o,d> >] = c;
      };

   {City} orig[p in Product] = { c.o | <p,c> in Routes };
   {City} dest[p in Product] = { c.d | <p,c> in Routes };

   {Connection} CP[p in Product] = { c | <p,c> in Routes };

   assert
      forall(p in Product)
        sum(o in orig[p]) supply[<p,o>] = sum(d in dest[p]) demand[<p,d>];

   var float+ trans[Routes];

   minimize
      sum(l in Routes) cost[l] * trans[l]
   subject to
      {
        forall(p in Product, o in orig[p])
           sum(<o,d> in CP[p]) trans[< p,<o,d> >] <= supply[<p,o>];

        forall(p in Product, d in dest[p])
           sum(<o,d> in CP[p]) trans[< p,<o,d> >] >= demand[<p,d>];

        forall(c in Connections)
```

```
            sum(<p,c> in Routes) trans[<p,c>] <= limit;
        };
```

References

[1] Peter Buneman, Leonid Libkin, Dan Suciu, Val Breazu-Tannen, and Limsoon Wong. Comprehension syntax. *ACM SIGMOD Record*, 23(1):87–96, March 1994. (Available online[7]).

[2] Rick Cattell, Douglas Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez, editors. *Object Data Standard—ODMG 3.0*. Morgan Kaufmann, January 2000.

[3] Bradford Chamberlain, Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An abstraction for expressing array computation. In *Proceedings of the SIGAPL/SIGPLAN International Conference on Array Programming Languages*, pages 41–49, August 1999. (Available online[8]).

[4] Bradford Chamberlain, Christopher Lewis, and Lawrence Snyder. A region-based approach for sparse parallel computing. Technical Report UW-CSE-98-11-01, University of Washington at Seattle (Computer Science), November 1998. (Available online[9]).

[5] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[6] Edgar F. Codd. Extending database relations to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, 1979.

[7] Edgar F. Codd. *The Relational Model for Database Management*. Addison-Wesley, 1990.

[8] Nachum Dershowitz. A taste of rewriting. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 199–228. Springer-Verlag, 1993. (Available online[10]).

[9] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):?–?, December 2000. (Available online[11]).

[10] Pascal van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

[11] Andreas Heuer and Marc H. Scholl. Principles of object-oriented query languages. In H.-J. Appelrath, editor, *Proceedings of the GI Conference on Database Systems for Office, Scientific, and Engineering Applications, Kaiserslautern, Germany*, pages 178–197. Springer-Verlag IFB 270, 1991. (Available online[12]).

[12] James Hoffman. Introduction to Structured Query Language. (Available online[13]), 1996–2001.

[7] `http://www.acm.org/sigs/sigmod/record/issues/9403/Comprehension.ps`

[8] `http://www.cs.washington.edu/homes/echris/papers/apl99.ps`

[9] `ftp://ftp.cs.washington.edu/tr/1998/11/UW-CSE-98-11-01.PS.Z`

[10] `http://www-sal.cs.uiuc.edu/~nachum/papers/taste-fixed.ps.gz`

[11] `http://lambda.uta.edu/tods00.ps.gz`

[12] `ftp://ftp.informatik.uni-konstanz.de/pub/dbis/Publications/HS:BTW91.ps.gz`

[13] `http://w3.one.net/~jhoffman/sqltut.htm`

[13] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, UK, 1970. Reprinted in *Automatic Reasoning*, 2, Springer-Verlag, pp. 342–276 (1983).

[14] Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Canada*, pages 228–239, May 1996. (Available online[14]).

[15] Frédéric Paulin. OPL language syntax critique and proposal for improvement. (Available online[15]), April 2001. ILOG internal access only.

[16] Scott L. Vandenberg. *Algebras for Object-Oriented Query Languages*. PhD thesis, University of Wisconsin at Madison (Computer Sciences), 1993. (Available online[16]).

[17] Limsoon Wong. *Querying Nested Collections*. PhD thesis, University of Pennsylvania (Computer and Information Science), 1994. (Available online[17]).

[14] http://www.cs.toronto.edu/~libkin/papers/sigmod96a.ps.gz
[15] file:/nfs/opl/matisse/dev/opl_syntax_critic.html
[16] ftp://ftp.cs.wisc.edu/pub/tech-reports/reports/1993/tr1161.ps.Z
[17] ftp://ftp.cis.upenn.edu/pub/ircs/tr/94-09.ps.Z