Optimizing Queries with Object Updates

Leonidas Fegaras

Department of Computer Science and Engineering
The University of Texas at Arlington
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015
email: *fegaras@cse.uta.edu*

Abstract

Object-oriented databases (OODBs) provide powerful data abstractions and modeling facilities but they usually lack a suitable framework for query processing and optimization. Even though there is an increasing number of recent proposals on OODB query optimization, only few of them are actually focused on query optimization in the presence of object identity and destructive updates, features often supported by most realistic OODB languages. This paper presents a formal framework for optimizing object-oriented queries in the presence of side effects. These queries may contain object updates at any place and in any form. We present a language extension to the monoid comprehension calculus to express these object-oriented features and we give a formal meaning to these extensions. Our method is based on denotational semantics, which is often used to give a formal meaning to imperative programming languages. The semantics of our language extensions is expressed in terms of our monoid calculus, without the need of any fundamental change to our basic framework. Our method not only maintains referential transparency, which allows us to do meaningful query optimization, but it is also practical for optimizing OODB queries since it allows the same optimization techniques applied to regular queries to be used with minimal changes for OODB queries with updates.

1 Introduction

One of the key factors for OODB systems to successfully compete with relational systems as well as to meet the performance requirements of many non-traditional applications is the development of an effective query optimizer. Even though there are many aspects to the OODB query optimization problem that can benefit from the, already successful, relational query optimization research, there many key features of OODB languages that make this problem unique and hard to solve. These features include object identity, methods, encapsulation, user-defined type constructors, large multimedia objects, multiple collection types, arbitrary nesting of collections, and nesting of query expressions.

There is an increasing number of recent proposals on OODB query optimization. Some of them are focused on handling nested collections [OW92, Col89], others on converting path expressions into joins [KM90, CD92], others on unnesting nested queries [CM95b, CM95a], while others are focused on handling encapsulation and methods [DGK$^+$91]. However, there very few proposals on query optimization in the presence of object identity and destructive updates, features often supported by most realistic OODB languages.

In earlier work [FM98, FM95b, FM95a], we proposed an effective framework with a solid theoretical basis for optimizing OODB query languages. Our calculus, called the *monoid comprehension calculus*, has already been shown to capture most features of ODMG OQL [Cat94] and is a good basis for expressing various optimization algorithms concisely, including query unnesting [Feg98] and translation of path expressions into joins [Feg97]. In this paper, we extend our framework to handle object identity and object updates.

1.1 Object Identity Complicates Query Optimization

Object-oriented programming is based on side-effects, that is, on the modification of the object store. Even though modern OODBs provide declarative query languages for associative access of data, queries in those languages are allowed to invoke any method, including those that perform side effects. Object creation itself, which is very common in OODB queries, is a side effect since it inserts a new object in a class extent. Consider for example the following OQL query:

```
select Person(e.name,e.address) from e in Employees
```

which creates a new person from each employee. Even though this query seems to be free of side effects at a first glance, it is not: it modifies the extent of the class Person by inserting a new person. If this query were a part of another query and this other query were scanning the extent of the class Person, this extent would have to be modified accordingly before it is used in the outer query. Therefore, the semantics of the above query must reflect the fact that the Person extent is modified during the execution of the query. Failure to do so may result to incorrect optimizations, which may lead to invalid execution plans.

The problem of assigning semantics to object-oriented queries becomes even worse if we allow object state modifications in arbitrary places in a query, like most OODB languages do. For example, the OQL query

```
select (e.salary := e.salary*1.08)
 from  e in Employees
where  e.salary>50000
```

gives an 8% raise to all employees earning over $50000. The semantics of this query should reflect the fact that a salary is modified at each iteration.

The situation where queries are mixed freely with updates occurs more frequently in OODB languages, such as O++ [AG89], that support set iteration embedded in a computational complete programming language. Even though these languages are beginning to disappear in favor to more declarative languages, such as OQL, there is a surge of interest to provide more computational power to existing declarative query languages without sacrificing performance. Consider for example the following O++ query (taken from [LD92]):

```
for (D of Divisions)
    for (E of Employees)
        suchthat (E→division==D) {
                totpay += ChristmasBonus + (E→bpay∗D→pftshare)/100;
                empcnt++; }
```

These types of queries allow any kind of C++ code inside a for-loop, including code that modifies the database. Earlier research on query optimization [LD92] has shown that queries in this form are very hard to optimize.

Another problem to consider when sets and bags are combined with side effects is that the results may be unpredictable due to commutativity. For example, in the following O++ query

```
for (e of Employees)
    x = e.salary
```

the value of x at the end of the execution of this program would depend on the way Employees are scanned. To understand the extent of this problem, consider the function f(n) which contains the assignment x:=n in its body and returns n. Is the value of x, after the execution of $\{\mathsf{f}(1), \mathsf{f}(2)\}$, 1 or 2? (Since $\{x, y\} = \{y, x\}$.) That is, is $\{\mathsf{f}(1), \mathsf{f}(2)\}$ equal to $\{\mathsf{f}(2), \mathsf{f}(1)\}$?

Given that side effects may appear at any place in a program, proving equivalences between OODB expressions becomes very hard. This makes the task of expressing and verifying optimization rules difficult to accomplish. For example, the well known transformation $x \bowtie y \rightarrow y \bowtie x$, for any terms x and y, is not valid any more since x and y may be queries that perform side effects and the order of execution of the side effects would be changed if this transformation were applied, thus changing the semantics of the program. One way to patch this error is to attach a guard to the above transformation rule to prevent its execution when both terms x and y contain side effects. Unfortunately, this approach is too conservative and it may miss some optimizations (e.g., when the side effects of x and y do not interfere to each other). Furthermore, there is a more fundamental problem with algebraic operations with side effects. For example, $x \bowtie y$ cannot appear in a valid translation of a declarative query, since a declarative query does not define an execution order.

Consequently, when optimizing a realistic OODB query language, we need to address the problem of object identity properly and handle the implicit or explicit side effects due to the use of object identity. It is also highly desirable to use the existing optimization techniques with minimal changes if possible. To do so, it is necessary to capture and handle object identity in the same framework as that for regular queries with no side effects. Unfortunately, such extensions are very difficult to incorporate to an existing optimization framework. To understand the level of difficulty, consider the following equality predicate:

```
Person("Smith","Park Av") = Person("Smith","Park Av")
```

where = is OID equality. This predicate must be evaluated to false since the left object has a different identity than the right one. On the other hand, given a function g(x) that computes the predicate x=x, the function call g(Person("Smith","Park Av")) should return true. But if we unfold the call to g, we get the previous false expression. Consequently, substituting the body of a function definition for a function call is not a valid transformation any more.

Our goal is to give a formal meaning to OODB queries with side effects, and more importantly, to provide an equational theory that allows us to do meaningful query optimization. It is highly desirable for this theory to be seamlessly incorporated into the monoid comprehension calculus, possibly by discovering a new monoid that captures the meaning of object identity. Another goal is, whenever there are no object updates in a query, we would like this query be treated in the same way as it is currently treated by our basic optimizer without the object extensions.

1.2 Our Approach

This paper presents a framework that incorporates impure features, namely object identity, into the monoid comprehension calculus. According to our earlier discussion, it is important to give semantics to such extensions to preserve *referential transparency*. We have referential transparency when we are able to substitute equal subexpressions in a context of a larger expression to give equal results [Rea89]. If a query language lacks this property, then the transformation rules of a query optimizer would depend on the context of the expression on which they apply.

Researchers in programming languages often use formal methods, such as denotational semantics, to solve such problems. In the denotational semantics approach, the impure features of a language can be captured by passing the state (here the object store) through all operations in a program. If a piece of a program does not update the state, then the state is propagated as is; otherwise it is *modified* to reflect the updates. When we say "the state is modified", we mean that a new copy of the state is created. This approach may become quite inefficient: for each destructive update, no matter how small it is, a new object store (i.e., an entire database) must be created. Obviously, this technique is unacceptable for most database applications. There is a solution to this problem. We can allow the state to be manipulated by a small number of primitives that not only preserve referential transparency, but have an efficient implementation as well. More specifically, even though these primitives are defined in a purely functional way, their implementations perform destructive updates to the state. That way we derive efficient programs and, more importantly, we maintain referential transparency, which allows us to do meaningful query optimization.

But there is a catch here: this solution works only if the state is *single-threaded* [Sch85]. Roughly speaking, a state is single-threaded through a program if this program does not undo any state modification at any point, that is, if there is a strict sequencing of all the state updates in the program. In that case, the state can be replaced by access rights to a single global variable and the state operations can be made to cause side effects while preserving operational properties. The following is an example of a non single-threaded program:

```
x := 1;
assume x:=2 in y := x+1;
```

The statement **assume** S1 **in** S2 executes the statement S1 locally. That is, the state modifications in S1 are used in S2 exclusively and then are discarded. Thus, the value of x and y after the completion of this program are 1 and 3 respectively (since the binding x:=2 is discarded after it is used in y:=x+1). This statement requires a local copy of the state during execution (probably in the form of a stack of states to handle nested **assume** statements), since it needs to backtrack to the previous state after the completion of execution. A rollback during a database transaction is another example of a non single-threaded operation.

There are two common ways to guarantee single-threadedness. The first is to allow any state manipulation in the language but detect violation of single-threadedness by performing a semantic analysis, i.e., a kind of abstract interpretation [Sch85], such as using a linear type system [SS95], to detect these violations during type-checking. The second approach, which we adopt in our framework, is to restrict the syntax of the language in such a way that the state is guaranteed to always be single-threaded.

There is another serious problem with the above mentioned denotational semantics approach: to pass the state through the operations of a program, we need to sequentialize all operations. This restriction is not a good idea for commutative operations, since we may miss some optimization opportunities. For example, for $R = \{1,2\}$ and r ranging over R, the assignment $x := r$ can be evaluated during the scanning of R in two ways; one will set x to 1 at the end and the other to 2, depending on the way

R is scanned. Both solutions are valid and should be considered by the optimizer. We address this problem by generating all possible solutions generated by these alternatives at a first stage. It is up to the optimizer to select the best one at the end (by performing a cost analysis). Even though this approach may generate an exponential number of solutions when applied to constant data, in practice it does not do so, provided that the query does not contain a large number of union operations. Each alternative solution corresponds to a typically different final database state. At the end, only one solution is chosen by the optimizer. So there is a "collection semantics" for queries – for each query there is a collection of possible correct answers. We decided to consider all solutions instead of reporting an error when more than one solution exists because most useful programs fall into this category. Considering all alternatives during query optimization is necessary for proving program equivalences (such as proving that $\{\mathsf{f}(1), \mathsf{f}(2)\}$ is equal to $\{\mathsf{f}(2), \mathsf{f}(1)\}$). Only at the plan generation phase, i.e., when optimization is completed, should we select an alternative.

Our framework is inspired by Ohori's work on representing object identity using monads [Oho90]. Our contribution is that we mix state transformation with sets and bags and that we apply this theory to a database query language that satisfies strong normalization properties. Normalization removes any unnecessary state transformation, thus making our approach practical for optimizing programs in our object-oriented calculus. The most important contribution of our work is the development of a method to map programs in which state transformation cannot be removed by normalization into imperative loops, much in the same way one could express these programs using a regular imperative language, such as C. The resulting programs are as efficient as those written by hand.

The rest of the paper is organized as follows. Section 2 describes our earlier results on the monoid comprehension calculus. Section 3 describes our object extensions to the monoid calculus. Section 4 proposes a new monoid that captures object identity and side effects. Section 5 describes our framework for handling object identity using denotational semantics. Section 6 addresses some practical considerations when building an optimizer based on our framework. Section 7 presents a prototype implementation of our framework. Finally, Section 8 extends our framework to capture database updates and discusses how this theory can be applied to solve the view maintenance problem.

2 Background: The Monoid Comprehension Calculus

This section summarizes our earlier work on the monoid calculus. A more formal treatment is presented elsewhere [FM98, FM95b, FM95a].

The monoid calculus is based on the concept of *monoids* from abstract algebra. A monoid of type T is a pair $(\oplus, Z_\oplus)$, where $\oplus$ is an associative function of type $T \times T \to T$ (i.e., a binary function that takes two values T and returns a value T), called the *accumulator* or the *merge* function of this monoid, and $Z_\oplus$ of type T, called the *zero* element of the monoid, is the left and right identity of $\oplus$. That is, the zero element satisfies $Z_\oplus \oplus x = x \oplus Z_\oplus = x$ for every x. Since the accumulator function uniquely identifies a monoid, we will often use the accumulator name as the monoid name. Examples of monoids include $(\cup, \{\,\})$ for sets, $(\uplus, \{\!\{\}\!\})$ for bags, $(+\!\!+, [\,])$ for lists, $(+, 0)$, $(*, 1)$, and $(max, 0)$ for integers, and $(\vee, \text{false})$ and $(\wedge, \text{true})$ for booleans. The monoids for integers and booleans are called *primitive monoids* because they construct values of a primitive type. The set, bag, and list monoids are called *collection monoids*. Each collection monoid $(\oplus, Z_\oplus)$ requires the additional definition of a *unit function*, $U_\oplus$, which, along with merge and zero, allows us the construction of all possible values of this type. For example, the unit function for the set monoid is $\lambda x.\,\{x\}$, that is, it takes a value x as input and constructs the singleton set $\{x\}$ as output. All but the list monoid are commutative, i.e., they satisfy $x \oplus y = y \oplus x$ for every x and y. In addition, some of them ($\cup$, $\wedge$, $\vee$, and max) are idempotent, i.e., they satisfy $x \oplus x = x$ for every x.

A *monoid comprehension* over the monoid $\oplus$ takes the form $\oplus\{\!\{\, e \,|\, \overline{r} \,\}\!\}$. Expression e is called the *head* of the comprehension. Each term r_i in the term sequence $\overline{r} = r_1, \ldots, r_n$, for $n \geq 0$, is called a *qualifier*, and is either a *generator* of the form $v \leftarrow e'$, where v is a *range variable* and e' is an expression (the generator domain) that constructs a collection, or a *filter* p, where p is a predicate. We will use the shorthand $\{\!\{\, e \,|\, \overline{r} \,\}\!\}$ to denote the set comprehension $\cup\{\!\{\, e \,|\, \overline{r} \,\}\!\}$.

A monoid comprehension is defined by the following reduction rules: ($\otimes$ is a collection monoid, possibly different than $\oplus$)

$$\oplus\{\!\{\, e \,|\, \,\}\!\} \quad \to \quad \begin{cases} U_\oplus(e) & \text{if } \oplus \text{ is a collection monoid} \\ e & \text{otherwise} \end{cases} \tag{D1}$$

$$\oplus\{\!\{\, e \,|\, \text{false}, \overline{r} \,\}\!\} \quad \to \quad Z_\oplus \tag{D2}$$

$$\oplus\{\!\{\, e \,|\, \text{true}, \overline{r} \,\}\!\} \quad \to \quad \oplus\{\!\{\, e \,|\, \overline{r} \,\}\!\} \tag{D3}$$

$$\oplus\{\, e \mid v \leftarrow Z_{\otimes},\, \bar{r} \,\} \quad \rightarrow \quad Z_{\oplus} \tag{D4}$$

$$\oplus\{\, e \mid v \leftarrow U_{\otimes}(e'),\, \bar{r} \,\} \quad \rightarrow \quad \textbf{let}\, v = e' \,\textbf{in}\, \oplus\{\, e \mid \bar{r} \,\} \tag{D5}$$

$$\oplus\{\, e \mid v \leftarrow (e_1 \otimes e_2),\, \bar{r} \,\} \quad \rightarrow \quad (\oplus\{\, e \mid v \leftarrow e_1,\, \bar{r} \,\}) \,\oplus\, (\oplus\{\, e \mid v \leftarrow e_2,\, \bar{r} \,\}) \tag{D6}$$

Rules (D2) and (D3) reduce a comprehension in which the leftmost qualifier is a filter, while Rules (D4) through (D6) reduce a comprehension in which the leftmost qualifier is a generator. The let-statement in (D5) binds v to e' and uses this binding in every free occurrence of v in $\oplus\{\, e \mid \bar{r} \,\}$.

The calculus has a semantic well-formedness requirement that a comprehension be over an idempotent or commutative monoid if any of its generators are over idempotent or commutative monoids. For example, $+\!\!+\{\, x \mid x \leftarrow \{1, 2\} \,\}$ is not a valid monoid comprehension, since it maps a set monoid (which is both commutative and idempotent) to a list monoid (which is neither commutative nor idempotent), while $+\{\, x \mid x \leftarrow \{\!\{1, 2\}\!\} \,\}$ is valid (since both $\uplus$ and $+$ are commutative). This requirement can be easily checked during compile time.

When restricted to sets, monoid comprehensions are equivalent to set monad comprehensions [BLS$^+$94], which capture precisely the nested relational algebra [FM95b]. Most OQL expressions have a direct translation into the monoid calculus. For example, the OQL query

```
select distinct hotel.price
 from  hotel in ( select h from c in Cities, h in c.hotels
                  where  c.name = "Arlington" )
where  exists r in hotel.rooms: r.bed_num = 3
  and  hotel.name in ( select t.name from s in States, t in s.attractions
                       where  s.name = "Texas" );
```

is translated into the following comprehension:

```
{ hotel.price  |  hotel ← { h  |  c ← Cities, h ← c.hotels, c.name="Arlington" },
                  ∨{ r.bed_num=3  |  r ← hotel.rooms },
                  ∨{ e=hotel.name  |  e ← { t.name  |  s ← States, t ← s.attractions,
                                                      s.name="Texas" } } }
```

We use the shorthand $x \equiv u$ to represent the binding of the variable x with the value u. The meaning of this construct is given by the following reduction:

$$\oplus\{\, e \mid \bar{r},\, x \equiv u,\, \bar{s} \,\} \longrightarrow \oplus\{\, e[u/x] \mid \bar{r},\, \bar{s}[u/x] \,\} \tag{N1}$$

where $e[u/x]$ is the expression e with u substituted for all the free occurrences of x (i.e., $e[u/x]$ is equivalent to $\textbf{let}\, x = u \,\textbf{in}\, e$). In addition, as a syntactic sugar, we allow irrefutable patterns in place of lambda variables, range variables, and variables in bindings. Patterns like these can be compiled away using standard pattern decomposition techniques [PJ87]. For example, $\{\, x + y \mid (x, (y, z)) \leftarrow A,\, z = 3 \,\}$ is equivalent to $\{\, a.\text{fst} + a.\text{snd.fst} \mid a \leftarrow A,\, a.\text{snd.snd} = 3 \,\}$, where fst/snd retrieves the first/second element of a pair. Another example is $\lambda(x, (y, z)).x + y + z$, which is a function that takes three parameters and returns their sum. It is equivalent to $\lambda a.\ a.\text{fst} + a.\text{snd.fst} + a.\text{snd.snd}$.

The monoid calculus can be put into a canonical form by an efficient rewrite algorithm, called the *normalization algorithm*. The evaluation of these canonical forms generally produces fewer intermediate data structures than the initial unnormalized programs. Moreover, the normalization algorithm improves program performance in many cases. It generalizes many optimization techniques already used in relational algebra, such as fusing two selections into one selection. The following are the most important rules of the normalization algorithm:

$$(\lambda v.e_1)\, e_2 \quad \longrightarrow \quad e_1[e_2/v] \qquad \text{beta reduction} \tag{N2}$$

$$\oplus\{\, e \mid \bar{q},\, v \leftarrow \{\, e' \mid \bar{r} \,\},\, \bar{s} \,\} \quad \longrightarrow \quad \oplus\{\, e \mid \bar{q},\, \bar{r},\, v \equiv e',\, \bar{s} \,\} \tag{N3}$$

$$\oplus\{\, e \mid \bar{q},\, \vee\{\, pred \mid \bar{r} \,\},\, \bar{s} \,\} \quad \longrightarrow \quad \oplus\{\, e \mid \bar{q},\, \bar{r},\, pred,\, \bar{s} \,\} \qquad \text{for idempotent } \oplus \tag{N4}$$

The soundness of the normalization rules can be proved using the definition of the monoid comprehension [FM98]. Rule (N3) flattens a comprehension that contains a generator whose domain is another comprehension (it may require variable renaming to avoid name conflicts). Rule (N4) unnests an existential quantification.

For example, the previous OQL query is normalized into:

```
{ h.price  |  c ← Cities, h ← c.hotels, r ← h.rooms, s ← States, t ← s.attractions,
              c.name="Arlington", r.bed_num=3, s.name="Texas", t.name=h.name }
```

by applying Rule (N3) to unnest the two inner set comprehensions and Rule (N4) to unnest the two existential quantifications.

3 The Object Monoid Calculus

In this section, the monoid calculus is extended to capture object identity. The extended calculus is called the *object monoid calculus*. For example, one valid object-oriented comprehension is

$$\mathbin{+\!\!+}\{\, !x \mid x \leftarrow [\text{new}(1), \text{new}(2)],\ x := !x + 1 \,\}$$

which first creates a list containing two new objects (new(1) and new(2)). Then, variable x ranges over this list and the state of x is incremented by one (by $x := !x{+}1$). Here x is a reference to the object x while $!x$ returns the state of the object x. The result of this computation is the list $[2, 3]$. An object-oriented comprehension is translated into a *state transformer* that propagates the object heap (which contains bindings from object identities to object states) through all operations in an expression, and changes it only when there is an operation that creates a new object or modifies the state of an object. This translation captures precisely the semantics of object identity without the need of extending the base model. It also provides an equational theory that allows us to do valid optimizations for object-oriented queries.

We introduce a new type constructor obj(T) that captures all objects with states represented by values of type T. In addition, we extend the monoid calculus with the following polymorphic operations [Oho90] (in the style of SML [Pau91]):

- new, of type $T \to \text{obj}(T)$. Operation new(s) creates a new object with state s;
- !, of type $\text{obj}(T) \to T$. Operation $!e$ dereferences the object e (it returns the state of e);
- :=, of type $\text{obj}(T) \to T \to \text{bool}$. Operation $e := s$ changes the state of the object e to s and returns true.

Many object-oriented languages have different ways of constructing and manipulating objects. For example, OQL uses object constructors to create objects and does not require any explicit object dereferencing operator. These language features can be easily expressed in terms of the primitives mentioned above. When giving formal semantics, our primitives are a better choice since they do not deal with details about object classes, inheritance, etc. When optimizing a real object-oriented language, though, these "details" should be addressed properly.

Our object-oriented operators may appear at any place in a monoid comprehension. The following are examples of comprehensions with object operations (called object comprehensions): (Recall that $v \equiv e$ defines the new variable name v to be a synonym of the value e while $e_1 := e_2$ changes the state of the object whose OID is equal to the value of e_1 into the result of e_2.)

1)	$\text{new}(1) = \text{new}(1)$	$\to$	false
2)	$!\text{new}(1) = !\text{new}(1)$	$\to$	true
3)	$\vee\{\, x = y \mid x \equiv \text{new}(1),\ y \equiv x,\ y := 2 \,\}$	$\to$	true
4)	$+\{\, !x \mid x \equiv \text{new}(1),\ y \equiv x,\ y := 2 \,\}$	$\to$	2
5)	$\{\, e \mid x \equiv \text{new}([\,]),\ x := [1,2],\ e \leftarrow !x \,\}$	$\to$	$\{1, 2\}$
6)	$\mathbin{+\!\!+}\{\, !x \mid x \leftarrow [\text{new}(1), \text{new}(2)],\ x := !x + 1 \,\}$	$\to$	$[2, 3]$
7)	$max\{\, !x \mid x \equiv \text{new}(0),\ e \leftarrow [1,2,3,4],\ x := !x + 1 \,\}$	$\to$	4
8)	$\mathbin{+\!\!+}\{\, !x \mid x \equiv \text{new}(0),\ e \leftarrow [1,2,3,4],\ x := !x + e \,\}$	$\to$	$[1, 3, 6, 10]$
9)	$max\{\, !x \mid x \equiv \text{new}(0),\ e \leftarrow \{1,2,2,3\},\ x := !x + 1 \,\}$	$\to$	3
10)	$\{\, !x \mid x \equiv \text{new}(0),\ e \leftarrow \{1,2,2,3\},\ x := !x + e \,\}$	$\to$	$\{1, 3, 6\}$

The first example indicates that different objects are distinct while the second example indicates that objects may have equal states. The ninth example computes the cardinality of the set $\{1, 2, 2, 3\} = \{1, 2, 3\}$ and indicates that duplicates in a set do not count. The last example is the most interesting one: since there is no order in the set $\{1, 2, 2, 3\}$, there are as many results as the permutations of the set (namely, $\{1, 3, 6\}$, $\{1, 4, 6\}$, $\{2, 3, 6\}$, $\{2, 5, 6\}$, $\{3, 4, 6\}$ and $\{3, 5, 6\}$). We consider all these results valid but the optimizer will construct a plan at the end that generates one result only. A more practical example is the query

$$+\{\, 1 \mid e \leftarrow \text{Employees},\ !e.\mathit{department} := !(!e.\mathit{manager}).\mathit{department} \,\}$$

which sets the department of each employee to be the department of the employee's manager.

4 The State Transformation Monoid

One way of handling side effects in denotational semantics is to map terms that compute values of type T into functions of type $S \rightarrow (T \times S)$, where S is the type of the state in which all side effects take place. That is, a term of type T is mapped into a function that takes some initial state s_0 of type S as input and generates a value of type T and a new state s_1. In denotational semantics, these functions of type $S \rightarrow (T \times S)$ are called state transformers [Wad92, Wad90]. If a term performs side effects, the state transformer maps s_0 into a different state s_1 to reflect these changes. Otherwise, the state remains unchanged. For example, the constant integer, 3, is mapped into the state transformer $\lambda s.(3, s)$ which propagates the state as is. Note that not only the new state, but the computed value as well, may depend on the input state. That way, side effects are captured as pure functions that map states into new states.

Unfortunately, if we add side effects to our calculus, our programs may have multiple interpretations, mainly due to the commutativity of monoids, which results to non-determinacy. It is highly desirable to capture all these interpretations and let the optimizer select the best one at the end. To handle this type of non-determinism in a functional way, given an input state, our state transformer must be able to return multiple values and multiple states, or in other words, it must be able to return multiple value-states pairs. Consequently, our state transformer should be of type $S \rightarrow \mathrm{set}(T \times S)$ to capture all possible interpretations of a program.

Definition 1 (State Transformer) *The* state transformer $\Phi(T)$ *of a type* T *and a state type* S *is the type* $S \rightarrow \mathrm{set}(T \times S)$.

As we will show shortly, given a monoid $\oplus$ for a type T, we can always define a primitive monoid, $\boxed{\oplus}$, for the state transformer $\Phi(T)$. In contrast to the monoids described earlier, this monoid must be a higher-order monoid, i.e., instances of this monoid are functions.

The definition of $\boxed{\oplus}$ described below is very important for proving the correctness of various transformation rules. It can be safely skipped if the reader is not interested in such proofs. We will first present a simple definition that works well for non-commutative, non-idempotent, monoids and then we will extent it to capture all monoids.

Definition 2 (State Transformation Monoid) *The* state transformation monoid *of a monoid* $(\oplus, Z_\oplus)$ *is the primitive monoid* $(\boxed{\oplus}, Z_{\boxed{\oplus}})$*, defined as follows:*

$$\begin{aligned} Z_{\boxed{\oplus}}\, s &= \{(Z_\oplus, s)\} \\ (\phi_1 \boxed{\oplus} \phi_2)\, s &= \{\!\!\{\, (v_1 \oplus v_2, s_2) \mid (v_1, s_1) \leftarrow \phi_1(s),\ (v_2, s_2) \leftarrow \phi_2(s_1) \,\}\!\!\} \end{aligned}$$

That is, $Z_{\boxed{\oplus}}$ is a function that, when applied to a state s of type S, it constructs a value $\{(Z_\oplus, s)\}$. The merge function $\boxed{\oplus}$ propagates the state from the first state transformer to the second and merges the resulting values using $\oplus$. It is easy to prove that for $n > 0$:

$$\begin{aligned} &(\phi_1 \boxed{\oplus} \phi_2 \boxed{\oplus} \cdots \boxed{\oplus} \phi_n)\, s \\ &= \{\!\!\{\, (v_1 \oplus v_2 \oplus \cdots \oplus v_n, s_2) \mid (v_1, s_1) \leftarrow \phi_1(s),\ (v_2, s_2) \leftarrow \phi_2(s_1), \\ &\qquad\qquad \ldots, (v_n, s_n) \leftarrow \phi_{n-1}(s_{n-1}) \,\}\!\!\} \end{aligned}$$

State monoid comprehensions are simply monoid comprehensions over a state transformation monoid. For example,

$$\begin{aligned} &\boxed{+}\{\!\!\{\, \lambda s.\, \{(v, s)\} \mid v \leftarrow [1, 2, 3] \,\}\!\!\}\, s_0 \\ &= ((\lambda s.\, \{(1, s)\}) \boxed{+} (\lambda s.\, \{(2, s)\}) \boxed{+} (\lambda s.\, \{(3, s)\}))\, s_0 \\ &= \{\!\!\{\, (v_1 + v_2 + v_3, s_3) \mid (v_1, s_1) \leftarrow \{(1, s_0)\},\ (v_2, s_2) \leftarrow \{(2, s_1)\}, \\ &\qquad\qquad (v_3, s_3) \leftarrow \{(3, s_2)\} \,\}\!\!\} \\ &= \{(1 + 2 + 3, s_0)\} = \{(6, s_0)\} \end{aligned}$$

The state can be of any type. Suppose that the state is an integer that counts list elements. Then the following state comprehension increments each element of the list $[1, 2, 3]$ and uses the state to count the list elements:

$$\begin{aligned} &\boxed{++}\{\!\!\{\, \lambda s.\, \{([v + 1], s + 1)\} \mid v \leftarrow [1, 2, 3] \,\}\!\!\}\, 0 \\ &= ((\lambda s.\, \{([2], s + 1)\}) \boxed{++} (\lambda s.\, \{([3], s + 1)\}) \boxed{++} (\lambda s.\, \{([4], s + 1)\}))\, 0 \\ &= \{\!\!\{\, ([v_1, v_2, v_3], s_3) \mid (v_1, s_1) \leftarrow \{([2], 0 + 1)\},\ (v_2, s_2) \leftarrow \{([3], s_1 + 1)\}, \\ &\qquad\qquad (v_3, s_3) \leftarrow \{([4], s_2 + 1)\} \,\}\!\!\} \\ &= \{([2, 3, 4], 3)\} \end{aligned}$$

For the state transformer monoid $\boxed{\oplus}$ to be effective, it must have the same properties as the monoid $\oplus$. Otherwise, it may introduce semantic inconsistencies. That is, if $\oplus$ is a commutative or idempotent, so must be $\boxed{\oplus}$. To capture this property, we redefine $\boxed{\oplus}$ to behave in the same way as $\oplus$:

$$(\phi_1 \boxed{\oplus} \phi_2)\, s \quad = \quad \pmb{\{}\, (v_1 \oplus v_2, s_2) \,|\, (v_1, s_1) \leftarrow \phi_1(s),\ (v_2, s_2) \leftarrow \mathcal{F}(\phi_2) \,\pmb{\}} \cup \mathcal{G}$$

where $\mathcal{F}$ and $\mathcal{G}$ are defined as follows:

$$\mathcal{G} \quad = \quad \begin{cases} \pmb{\{}\, (v_1 \oplus v_2, s_2) \,|\, (v_1, s_1) \leftarrow \phi_2(s),\ (v_2, s_2) \leftarrow \mathcal{F}(\phi_1) \,\pmb{\}} & \text{if } \oplus \text{ is commutative} \\ \{\,\} & \text{otherwise} \end{cases}$$

$$\mathcal{F}(\phi) \quad = \quad \begin{cases} \textbf{if } (v_1, s_1) \in \phi(s) \textbf{ then } \{(Z_\oplus, s_1)\} \textbf{ else } \phi(s_1) & \text{if } \oplus \text{ is idempotent} \\ \phi(s_1) & \text{otherwise} \end{cases}$$

where $x \in R \equiv \vee\pmb{\{}\, x = z \,|\, z \leftarrow R \,\pmb{\}}$. That is, if $\oplus$ is commutative, then $\phi_1 \boxed{\oplus} \phi_2$ has two interpretations: one propagates the state from ϕ_1 to ϕ_2 and the other from ϕ_2 to ϕ_1 (this is the contribution of the factor $\mathcal{G}$). If $\oplus$ is idempotent, then all elements of x are removed from y when $x \oplus y$ is evaluated. For example, for an integer state that counts set elements we have:

$$\begin{aligned} & (\lambda s.\{(\{1\}, s+1)\}) \boxed{\cup} (\lambda s.\{(\{1\}, s+1)\})\, s \\ = \ & \pmb{\{}\, (v_1 \cup v_2, s_2) \,|\, (v_1, s_1) \leftarrow \{(\{1\}, s+1)\}, \\ & \qquad (v_2, s_2) \leftarrow (\textbf{if } (v_1, s_1) \in \{(\{1\}, s+1)\} \\ & \qquad\qquad \textbf{then } \{(Z_\oplus, s_1)\} \\ & \qquad\qquad \textbf{else } \{(\{1\}, s_1+1)\}) \,\pmb{\}} \cup \mathcal{G} \\ = \ & \pmb{\{}\, (\{1\} \cup v_2, s_2) \,|\, (v_2, s_2) \leftarrow \{(Z_\oplus, s+1)\} \,\pmb{\}} \cup \mathcal{G} \\ = \ & \{(\{1\}, s+1)\} \cup \mathcal{G} \end{aligned}$$

where $\mathcal{G}$ propagates the state from right to left and it is equal to $\{(\{1\}, s+1)\}$. That is, the counter counts the list element 1 once, even though it appears twice.

We prove in the Appendix (Theorem 1) that, under these extensions, not only $\boxed{\oplus}$ is a valid monoid, but it is also compatible with the $\oplus$ monoid (i.e., if $\oplus$ is commutative and/or idempotent, then so is $\boxed{\oplus}$).

5 Capturing Object Identity

So far we have not discussed what the state type, S, should be. Indeed, S can be of any type. If we wished to capture database updates, for example, the state would have to be the entire database. Here, though, we are interested in capturing object identity. The state s of a state transformer that captures object identity can be viewed as a pair (L, n) [Oho90]. The value L, called the *object store*, maps objects of type T (i.e., instances of the type obj(T)) into values of type T. That is, it maps OIDs into object states. The integer n is a counter used for computing the next available OID.

There are four primitives to manipulate the state, which have the following types:

$$\begin{array}{lcl} \text{ref}^T & : & \text{int} \rightarrow \text{obj}(T) \\ \text{emptyStore} & : & \text{store} \\ \text{ext}^T & : & (\text{store} \times \text{obj}(T) \times T) \rightarrow \text{store} \\ \text{lookup}^T & : & (\text{store} \times \text{obj}(T)) \rightarrow T \end{array}$$

$\text{ref}^T(n)$ maps the integer n into an OID that references an object of type T, emptyStore is the initial object store value (without any objects), $\text{ext}^T(L, o, v)$ extends the object store L with the binding from the OID o to the state value v, and $\text{lookup}^T(L, o)$ accesses the object store L to retrieve the state of the object with OID o. For example,

$$\text{lookup}^{int}(\text{ext}^{int}(L, \text{ref}^{int}(100), 1), \text{ref}^{int}(100))$$

returns 1, the state of the object (of type obj(int)) with OID 100.

The abover primitives satisfy the following equivalences:

$$\begin{aligned} \text{ref}^T(n) = \text{ref}^T(m) \quad &\equiv \quad n = m && \text{(O1)} \\ \text{ext}^T(\text{ext}^T(L, x, v), x, u) \quad &\equiv \quad \text{ext}^T(L, x, u) && \text{(O2)} \\ \text{ext}^T(\text{ext}^T(L, x, v), y, u) \quad &\equiv \quad \text{ext}^T(\text{ext}^T(L, y, u), x, v) && \text{(O3)} \\ \text{lookup}^T(\text{ext}^T(L, x, v), x) \quad &\equiv \quad v && \text{(O4)} \\ \text{lookup}^T(\text{ext}^T(L, x, v), y) \quad &\equiv \quad \text{lookup}^T(L, y) \qquad \text{if } x \neq y && \text{(O5)} \end{aligned}$$

Figure 1 presents the denotational semantics of the most important constructs of the monoid calculus without the object extensions (i.e., without new, !, and :=). Without the object extensions, the state

$$
\begin{aligned}
\llbracket c \rrbracket\, s &= \{(c,s)\} && \text{(S1)}\\
\llbracket v \rrbracket\, s &= \{(v,s)\} && \text{(S2)}\\
\llbracket e_1 = e_2 \rrbracket\, s &= \{\, (v_1 = v_2, s_2) \mid (v_1,s_1) \leftarrow \llbracket e_1 \rrbracket\, s,\ (v_2,s_2) \leftarrow \llbracket e_2 \rrbracket\, s_1 \,\} && \text{(S3)}\\
\llbracket \langle A_1 = e_1 \ldots A_n = e_n \rangle \rrbracket\, s & && \text{(S4)}\\
&= \{\, (\langle A_1 = v_1 \ldots A_n = v_n \rangle, s_n) \mid (v_1,s_1) \leftarrow \llbracket e_1 \rrbracket\, s, \ldots,\ (v_n,s_n) \leftarrow \llbracket e_n \rrbracket\, s_{n-1} \,\} &&\\
\llbracket e.A_i \rrbracket\, s &= \{\, (v.A_i, s') \mid (v,s') \leftarrow \llbracket e \rrbracket\, s \,\} && \text{(S5)}\\
\llbracket \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rrbracket\, s & && \text{(S6)}\\
&= \{\, (v_2, s_2) \mid (v_1,s_1) \leftarrow \llbracket e_1 \rrbracket\, s,\ (v_2,s_2) \leftarrow \textbf{if } v_1 \textbf{ then } \llbracket e_2 \rrbracket\, s_1 \textbf{ else } \llbracket e_3 \rrbracket\, s_1 \,\} &&\\
\llbracket \lambda v.e \rrbracket\, s &= \{(\lambda v.\lambda s'.\llbracket e \rrbracket\, s', s)\} && \text{(S7)}\\
\llbracket e_1(e_2) \rrbracket\, s &= \{\, (v, s_3) \mid (f,s_1) \leftarrow \llbracket e_1 \rrbracket\, s,\ (x,s_2) \leftarrow \llbracket e_2 \rrbracket\, s_1,\ (v,s_3) \leftarrow (f(x)\, s_2) \,\} && \text{(S8)}\\
\llbracket Z_\oplus \rrbracket\, s &= \{(Z_\oplus, s)\} && \text{(S9)}\\
\llbracket U_\oplus(e) \rrbracket\, s &= \{\, (U_\oplus(v), s') \mid (v,s') \leftarrow \llbracket e \rrbracket\, s \,\} && \text{(S10)}\\
\llbracket e_1 \oplus e_2 \rrbracket\, s &= (\llbracket e_1 \rrbracket \boxed{\oplus} \llbracket e_2 \rrbracket)\, s && \text{(S11)}\\
\llbracket \oplus\{\, e \mid \,\} \rrbracket\, s &= \{\, (U_\oplus(v), s') \mid (v,s') \leftarrow \llbracket e \rrbracket\, s \,\} && \text{(S12)}\\
\llbracket \oplus\{\, e \mid pred,\ \bar{r} \,\} \rrbracket\, s &= \{\, (v, s_2) \mid (p,s_1) \leftarrow \llbracket pred \rrbracket\, s,\ p,\ (v,s_2) \leftarrow \llbracket \oplus\{\, e \mid \bar{r} \,\} \rrbracket\, s_1 \,\} && \text{(S13)}\\
\llbracket \oplus\{\, e \mid v \leftarrow u,\ \bar{r} \,\} \rrbracket\, s &= \{\, (v_2, s_2) \mid (v_1,s_1) \leftarrow \llbracket u \rrbracket\, s,\ (v_2,s_2) \leftarrow \boxed{\oplus}\{\, \llbracket \oplus\{\, e \mid \bar{r} \,\} \rrbracket \mid v \leftarrow v_1 \,\}\, s_1 \,\} && \text{(S14)}
\end{aligned}
$$

Figure 1: Denotational Semantics of the Monoid Calculus Using State Transformers

should be propagated as is, not changed. The semantics of the object extensions form a non-standard interpretation and is given later. In these equations, the semantic brackets, $\llbracket\,\rrbracket$, give the meaning of the syntax enclosed by the brackets in terms of the pure monoid comprehension calculus. The type of $\llbracket\,\rrbracket$ is $L_T \to \Phi(T)$, where L_T is the domain of all type-correct terms in the object monoid calculus that have a monotype T (a non-function type). In general, if e is of type t, then the type of $\llbracket e \rrbracket$, denoted by t^*, is defined as follows:

$$
\begin{aligned}
(t_1 \to t_2)^* &= \Phi(t_1 \to t_2^*) && \textit{for a function type}\\
t^* &= \Phi(t) && \textit{for a monotype } t
\end{aligned}
$$

Recall also that a state s of type S is a pair (L, n); but for convenience, we will only use the notation (L, n) whenever either of the components L or n needs to be accessed.

Rule (S7) in Figure 1 handles functional terms. For example, $\llbracket \lambda v.v \rrbracket$ has type $\Phi(T \to \Phi(T))$ and it is translated into $\{(\lambda v.\lambda s'.\{(v, s')\}, s)\}$ when applied to a state s. Rule (S8) assumes a call-by-value interpretation [Rea89]: e_2 in $e_1(e_2)$ is evaluated before e_1 is applied. Rules (S13) and (S14) translate monoid comprehensions. Rule (S14) uses a monoid comprehension over the monoid $\boxed{\oplus}$ to propagate the state through every element of the collection u. Notice that the comprehension head here is a state transformer and all these state transformers are merged using $\boxed{\oplus}$. This comprehension is valid for any type of collection u, since the monoid $\boxed{\oplus}$ is compatible with the monoid $\oplus$. This higher-order comprehension is necessary since the term u may modify the object store each time a new object is constructed. In most cases, though, the state is propagated but not changed. If it is not changed, the following rule can be applied to eliminate state propagation (the correctness of this rule is straightforward and is omitted):

$$
\boxed{\oplus}\{\, \lambda s.\{(e,s)\} \mid v \leftarrow u \,\}\, s \quad \to \quad \{(\oplus\{\, x \mid v \leftarrow u,\ x \leftarrow e \,\}, s)\} \qquad \text{(N11)}
$$

The following rules give the denotational semantics of the object extensions:

$$
\begin{aligned}
\llbracket \mathrm{new}(e) \rrbracket\, s &= \{\, (\mathrm{ref}^T(n), (\mathrm{ext}^T(L, \mathrm{ref}^T(n), a), n+1)) \mid (a, (L,n)) \leftarrow \llbracket e \rrbracket\, s \,\} && \text{(S15)}\\
\llbracket e_1 := e_2 \rrbracket\, s &= \{\, (\mathrm{true}, (\mathrm{ext}^T(L, o, v), n)) \mid (o, s') \leftarrow \llbracket e_1 \rrbracket\, s,\ (v, (L,n)) \leftarrow \llbracket e_2 \rrbracket\, s' \,\} && \text{(S16)}\\
\llbracket !e \rrbracket\, s &= \{\, (\mathrm{lookup}^T(L, v), (L,n)) \mid (v, (L,n)) \leftarrow \llbracket e \rrbracket\, s \,\} && \text{(S17)}
\end{aligned}
$$

The operation new(e) takes an available OID, n, and uses it as the OID of the new object with state e. In addition, the object store is extended with the binding from $\mathrm{ref}^T(n)$ to the state value. Rule (S16), instead of destructively changing the object store, extends the store with a new binding from the OID of the left part of := to the value of the right part. Rule (S17) simply looks up the object store for the requested OID.

The Appendix provides a proof of a theorem (Theorem 2) that indicates that, if we have no state modification operations in the calculus, the output state is the same as the input state and that the canonical form we derive after normalization is similar to the canonical form we get in the pure monoid calculus. This theorem basically guarantees that, even though state transformation sequentializes all operations, if a program does not perform any state modification, the normalization algorithm can remove all unnecessary state transformations.

The following are examples of translation and normalization of some terms in the object monoid calculus (where the state s is (L, n)):

$$\begin{aligned}
&[\![!x+1]\!]\,(L,n) &&\rightarrow \{(\mathrm{lookup}(L,x)+1,(L,n))\}\\
&[\![x := !x+1]\!]\,(L,n) &&\rightarrow \{(\mathrm{true},(\mathrm{ext}(L,x,\mathrm{lookup}(L,x)+1),n))\}\\
&[\![\mathrm{new}(1)]\!]\,(L,n) &&\rightarrow \{(\mathrm{ref}(n),(\mathrm{ext}(L,\mathrm{ref}(n),1),n+1))\}\\
&[\![+\{\!\!\{ !x \mid x \equiv \mathrm{new}(1),\ x := !x+1 \}\!\!\}]\!]\,(L,n) &&\rightarrow \{(2,(\mathrm{ext}(L,\mathrm{ref}(n),2),n+1))\}
\end{aligned}$$

A more interesting example is incrementing all elements of a set of integers (of type set(obj(int))):

$$\begin{aligned}
&[\![\{\!\!\{ !x \mid x \leftarrow R,\ x := !x+1 \}\!\!\}]\!]\,s\\
&\rightarrow \boxed{\cup}\{\!\!\{ \lambda(L,n).\{(\{\mathrm{lookup}(L,x)+1\},(\mathrm{ext}(L,x,\mathrm{lookup}(L,x)+1),n))\} \mid x \leftarrow R \}\!\!\}\,s
\end{aligned}$$

Set cardinality can be expressed with the help of a counter x:

$$\begin{aligned}
&[\![\mathit{max}\{\!\!\{ !x \mid x \equiv \mathrm{new}(0),\ a \leftarrow R,\ x := !x+1 \}\!\!\}]\!]\,(L,n)\\
&\rightarrow \boxed{\mathit{max}}\{\!\!\{ \lambda(L,n).\{(\mathrm{lookup}(L,\mathrm{ref}(n))+1,(\mathrm{ext}(L,\mathrm{ref}(n),\mathrm{lookup}(L,\mathrm{ref}(n))+1),n))\}\\
&\qquad\qquad \mid a \leftarrow R \}\!\!\}\\
&\quad (\mathrm{ext}(L,\mathrm{ref}(n),0),n+1)
\end{aligned}$$

We can now support bags and sets of objects without inconsistencies. For example, $\{\!\{\mathrm{new}(1), \mathrm{new}(2)\}\!\}$ is a valid expression and it is equal to $\{\!\{\mathrm{new}(2), \mathrm{new}(1)\}\!\}$:

$$\begin{aligned}
&[\![\{\!\{\mathrm{new}(1),\mathrm{new}(2)\}\!\}]\!]\,(L,n)\\
=\ &\{(\{\!\{\mathrm{ref}(n),\mathrm{ref}(n+1)\}\!\},\ (\mathrm{ext}(\mathrm{ext}(L,\mathrm{ref}(n),1),\mathrm{ref}(n+1),2),n+2)),\\
&\ (\{\!\{\mathrm{ref}(n+1),\mathrm{ref}(n)\}\!\},\ (\mathrm{ext}(\mathrm{ext}(L,\mathrm{ref}(n+1),1),\mathrm{ref}(n),2),n+2))\}\\
=\ &[\![\{\!\{\mathrm{new}(2),\mathrm{new}(1)\}\!\}]\!]\,(L,n)
\end{aligned}$$

Similarly, assignments can be freely moved inside set constructions:

$$\begin{aligned}
[\![\{\!\{x := 1,\ x := 2\}\!\}]\!]\,(L,n) &= \{(\{\!\{\mathrm{true},\ \mathrm{true}\}\!\},(\mathrm{ext}(L,x,2),n)),\\
&\qquad (\{\!\{\mathrm{true},\ \mathrm{true}\}\!\},(\mathrm{ext}(L,x,1),n))\}\\
&= [\![\{\!\{x := 2,\ x := 1\}\!\}]\!]\,(L,n)
\end{aligned}$$

6 Translating Object Comprehensions into Efficient Programs

We have seen in the previous section that object-oriented comprehensions can be expressed in term of the basic monoid comprehension calculus using the denotational semantics rules of Figure 1. The resulting programs are usually inefficient since they manipulate the state even when the state is not used at all. These inefficiencies can be reduced with the help of the normalization algorithm and the algebraic equalities for the object primitives (Rules (O1) through (O5)). In fact, most parts of the resulting programs can be normalized to first-order programs that look very similar to the programs one might write using the four object primitives directly. This section is focused on the efficient execution of the programs that cannot be reduced to first-order programs by the normalization algorithm.

When translating the object monoid calculus to the basic calculus we consider all possible alternatives due to the commutativity of some operations. This is absolutely necessary for proving program equalities. After the normalization is completed and the algebraic equalities have been used to check program equivalences, the optimizer can safely discard all but one alternative. The following function, choose, selects a random alternative. In practice the choice can be made with the help of a cost function. Given a program P in the object calculus and an initial state s_0, our system evaluates choose(normalize($[\![P]\!]\,s_0$)).

That is, P is first translated, then normalized, and finally an alternative is selected (which is a pair of a value and a state). The choose function is defined as follows:

$$\begin{array}{lcl}
\text{choose}(x \cup y) & = & \text{either choose}(x)\text{, if } x \neq \emptyset\text{, or choose}(y)\text{, if } y \neq \emptyset \\
\text{choose}(\{a\}) & = & a \\
\text{choose}(\{\!\{\, e \mid \,\}\!\}) & = & e \\
\text{choose}(\{\!\{\, e \mid v \leftarrow X,\ \bar{r} \,\}\!\}) & = & \textbf{let } v = \text{choose}(X) \textbf{ in } \text{choose}(\{\!\{\, e \mid \bar{r} \,\}\!\})
\end{array}$$

The rules in Figure 1 guarantee that there will always be at least one choice. The only case missing from the above rules is choosing an alternative from a state monoid comprehension. These are the state monoid comprehensions that cannot be removed by the normalization algorithm. Efficient implementation of such comprehensions is very crucial when considering system performance. The default implementation of a state monoid comprehension is a loop that creates a state transformer (i.e., a function) at each iteration and then composes these state transformers using the merge function of the state transformation monoid. This approach is obviously inefficient and we would like to find better algorithms to evaluate state comprehensions faster. One possible solution is to actually compile these comprehensions into loops with updates, like the ones found in imperative languages. In particular,

$$\text{choose}(\,\boxed{\oplus}\{\!\{\, \lambda s.\, e \mid v \leftarrow R \,\}\!\}\, s_0\,)$$

is translated into the following loop (in a Pascal-like syntax):

```
s := $s_0$;                     initialize the state
res := $Z_\oplus$;                 initialize the result value
for each $v$ in $R$ do
{     x := choose(e);           retrieve one of the possible value-state pairs
      res := res $\oplus$ x.fst;      update the result
      s := x.snd;               update the state
};
return (res,s);
```

For example, as we have previously shown, set cardinality is translated into the following state monoid comprehension:

$$\begin{array}{l}
\boxed{max}\{\!\{\, \lambda(L,n).\{(\text{lookup}(L,\text{ref}(n))+1,(\text{ext}(L,\text{ref}(n),\text{lookup}(L,\text{ref}(n))+1),n))\} \\
\quad\ \mid a \leftarrow R \,\}\!\}\ s_0
\end{array}$$

which is mapped into the following loop:

```
$(L,n)$ := $s_0$;
$res$ := 0;
for each $a$ in $R$ do
{     $res$ := $max(res, \text{lookup}(L,\text{ref}(n))+1)$;
      $(L,n)$ := $(\text{ext}(L,\text{ref}(n),\text{lookup}(L,\text{ref}(n))+1),n)$;
};
return $(res,(L,n))$;
```

Even though this loop has the right functionality, it is still inefficient since it manipulates the object store L at every step of the loop.

The resulting programs can be implemented efficiently if the store is a global array and the object primitives are programs that directly manipulate the global array. Rules (S15) and (S16) are single-threaded (since no object creation is undone at any point). Rule (S17) can enforce a single array pointer by fetching the state first (using lookup) and then by returning the pointer to L. Consequently, there will always be only one pointer to the store, and therefore, this store can be implemented as a global store and all updates as in-place (destructive) updates.

Any primitive operation on the object store can be done destructively. More specifically, let Store be the global array mentioned above whose domain of elements is of any type (e.g., Store can be defined as void[ ] in C). The object primitives can be implemented as follows:

$$\begin{array}{lcl}
\text{ref}^T(n) & \mapsto & n \\
\text{lookup}^T(s,x) & \mapsto & (s',\ \text{Store}[x]) \\
\text{ext}^T(s,x,v) & \mapsto & s',\ \text{Store}[x]\ :=\ v
\end{array}$$

where s' is the implementation of s and $(s_1,\ s_2,\ldots,s_n,\ e)$ evaluates the statements $s_1,\ldots,s_n$ in that order and returns e. For example, $\text{lookup}^T(\text{ext}^T(\text{ext}^T(s,x,a),y,b),z)$ is translated into

$$(\text{Store}[x]\ :=\ a,\ \text{Store}[y]\ :=\ b,\ \text{Store}[z])$$

Under this implementation, the resulting programs after state transformation can be evaluated as efficiently as real object-oriented programs.

For example, the previous loop that corresponds to set cardinality becomes:

```
n := $n_0$;
res := 0;
for each $a$ in $R$ do
{      res := max(res,Store[n]+1);
       Store[n] := Store[n]+1; n := n;
};
return (res,n);
```

if we use the global array implementation of the object primitives.

7 Implementation

We have already built a prototype implementation of our framework. The translations of the OODB queries shown in this paper were generated by this program. The source code is available at:

`http://www-cse.uta.edu/~fegaras/oid/`

The following examples illustrate the translation of five object queries by our system. The notation used in these examples is a little bit different than that used in our theoretical framework. Here block($s_1, \ldots, s_n, v$) executes the statements $s_1, \ldots, s_n$ in sequence and returns the value of v, loop(iterate(x, X),$s_1, \ldots, s_n$) executes the statements $s_1, \ldots, s_n$ for each value x of X, access(n) returns the value of Store[n], and update(n, v) evaluates Store[n] := v. Every object query of type T is translated into an expression of type $T \times (\text{void} \times \text{int})$. The T value is the returned value, the void value corresponds to the state which is ignored, and the int value is the new OID counter (we assume that the value of the OID counter before the execution of the query is equal to n). For example, if an object query e does not contain any object operation, the translation would be pair(e,pair(null,n)), where pair constructs a pair of two values and null is of type void.

```
> new(1)
= pair(n,pair(block(update(n,1),null),plus(n,1)))

> assign(x,plus(deref(x),1))
= pair(true,pair(block(update(x,plus(access(x),1)),null),n))

> compr(sum,deref(x),bind(x,new(1)),assign(x,plus(deref(x),1)))
= pair(2,pair(block(update(n,1),update(n,2),null),plus(n,1)))

> compr(max,deref(x),bind(x,new(0)),iterate(e,E),assign(x,plus(deref(x),e)))
= block(assign(s,pair(block(update(n,0),null),plus(n,1))),
        assign(res,0),
        loop(iterate(e,E),
             assign(res,max(res,plus(access(n),e))),
             assign(s,pair(block(update(n,plus(access(n),e)),null),snd(s)))),
        pair(res,s))

> compr(sum,1,iterate(e,Employees),
              gt(project(deref(e),salary),50000),
              assign(e,struct(bind(name,project(deref(e),name)),
                              bind(salary,times(project(access(e),salary),1.08)))))
= block(assign(s,pair(null,n)),
        assign(res,0),
        loop(iterate(e,Employees),
             assign(res,plus(res,if(gt(project(access(e),salary),50000),1,0))),
             assign(s,if(gt(project(access(e),salary),50000),
                         pair(block(update(e,struct(bind(name,project(access(e),name)),
                                          bind(salary,times(project(access(e),salary),1.08)))),
                                    null),
                              snd(s)),
                         s))),
        pair(res,s))
```

$$
\begin{array}{rcll}
[\![\oplus\{\, e \,|\, pred,\, \overline{r} \,\}]\!]\,\rho\, s & = & \{\, (v,s_2) \,|\, (p,s_1) \leftarrow [\![pred]\!]\,\rho\, s,\; p,\; (v,s_2) \leftarrow [\![\oplus\{\, e \,|\, \overline{r} \,\}]\!]\,\rho\, s_1 \,\} & \text{(S13)} \\
[\![\oplus\{\, e \,|\, v \leftarrow u,\, \overline{r} \,\}]\!]\,\rho\, s & = & \{\, (v_2,s_2) \,|\, (v_1,s_1) \leftarrow [\![u]\!]\,\rho\, s, & \text{(S14)} \\
 & & \qquad (v_2,s_2) \leftarrow \boxed{\oplus}\{\, [\![\oplus\{\, e \,|\, \overline{r} \,\}]\!]\,\rho[v_1/v] \,|\, v \leftarrow v_1 \,\}\, s_1 \,\} & \\
[\![\oplus\{\, e \,|\, path := u,\, \overline{r} \,\}]\!]\,\rho\, s & = & \{\, (v_2,s_2) \,|\, (v_1,s_1) \leftarrow [\![u]\!]\,\rho\, s, & \text{(S15)} \\
 & & \qquad (v_2,s_2) \leftarrow [\![\oplus\{\, e \,|\, \overline{r} \,\}]\!]\,\rho\, (\mathcal{P}[\![path]\!]\,\rho\, v_1\, s_1) \,\} &
\end{array}
$$

Figure 2: Denotational Semantics of Database Updates

The first query, new(1), assigns 1 to Store[n] (the store for the next available OID), sets n to $n+1$, and returns the old value of n. The second query executes Store[x] := Store[x] + 1, while the third query, which corresponds to $+\{\, !x \,|\, x \equiv \text{new}(1),\, x := !x+1 \,\}$, generates a block that contains both Store[n] := 1 (the old value) and Store[n] := 2 (the new value) that are executed in sequence. The fourth query generates a state monoid comprehension, which, in turn, is translated into a loop. It returns the sum of all elements in E. The last query gives an 8% raise to all employees earning over \$50000 and returns the number of employees who got this raise (for simplicity, we assume that each employee has a name and a salary only).

8 Database Updates and View Maintenance

The monoid state transformer can also be used for expressing destructive database updates in terms of the basic algebra. That way database updates and queries can be optimized in a single framework. Let *db* be the current database state and let *DB* be its type. Typically, *db* is the aggregation of all persistent objects in an application. Following the analysis of the previous section, we want to translate updates into the monoid algebra in such a way that the propagated state (i.e. the entire database) is single-threaded. Database updates can be captured using the state transformer

$$\Phi(T) \;=\; DB \rightarrow \text{set}(T \times DB)$$

which propagates, and occasionally changes, the entire database. That is, updates are captured as pure functions that map the existing database into a new database. To make this approach practical, we can define a set of update primitives to express updates, similar to the ones for object updates. These primitives, even though have a pure interpretation, they have an efficient implementation. This approach does not require any significant extension to the formal framework, and normalization and query optimization can be used as is to improve performance.

Database updates can be expressed using the following comprehension qualifiers [Feg]: qualifier $path := u$ destructively replaces the value stored at *path* by u, qualifier $path \mathrel{+}= u$ merges the singleton u with *path*, and qualifier $path \mathrel{-}= u$ deletes all elements of *path* equal to u. The $:=$ qualifier is the only fundamental construct since the other two can be defined as follows:

$$
\begin{array}{rcl}
path \mathrel{+}= u & \equiv & path := (U_{\oplus}(u) \oplus path) \\
path \mathrel{-}= u & \equiv & path := \oplus\{\, a \,|\, a \leftarrow path,\, a \neq u \,\}
\end{array}
$$

For example, the comprehension

```
+{ 1 | c ← db.cities, c.name="Arlington",
       c.hotels += ⟨ name="Hilton", rooms=100 ⟩,
       c.hotel_num += 1 }
```

inserts a new hotel in Arlington and increases the total number of hotels.

The denotational semantics of an expression e that may contain updates is $[\![e]\!]\,\rho\, s$, where ρ is a binding list that binds range variables and s is the current database state. Rules (S1) through (S13) in Figure 1 need to be slightly modified to include the binding list ρ: every $[\![\ldots]\!]\, s$ term is mapped into $[\![\ldots]\!]\,\rho\, s$. For example, Rule (S13) in Figure 1 is mapped into the Rule (S13) in Figure 2. Rule (S14) in Figure 1, though, is mapped into the Rule (S14) in Figure 2, which changes ρ to include the binding from v (the range variable) to v_1 (the generator domain). The binding list ρ is used in Rule (S15), which gives the semantics of an update qualifier.

Expression $\mathcal{P}[\![path]\!]\,\rho\,v\,s$ reconstructs the database state s by copying all its components except the one reached by *path*, which is replaced by v. It is defined as follows:

$$\begin{array}{rcl}
\mathcal{P}[\![v]\!]\,\rho\,e\,s & = & e \\
\mathcal{P}[\![v.path]\!]\,\rho\,e\,s & = & \mathcal{P}[\![\rho(v)]\!]\,\rho\,(\oplus\{\!\{\,\mathcal{P}[\![w.path]\!]\,\rho\,e\,w \mid w \leftarrow [s/db]\rho(v),\ w = v\,\}\!\})\,s \\
 & & \text{where } v \in \rho \text{ and } \rho(v) \text{ is of type } T_{\oplus}(t) \\
\mathcal{P}[\![u.A_i.path]\!]\,\rho\,e\,s & = & \langle A_1 = s.A_1, \ldots, A_i = \mathcal{P}[\![A_i.path]\!]\,\rho\,e\,(s.A_i), \ldots, A_n = s.A_n\rangle \\
 & & \text{where } s \text{ is of type } \langle A_1 : t_1, \ldots, A_n : t_n\rangle
\end{array}$$

where *path* is a possibly empty path (a sequence of projections). Expression $[s/db]\rho(v)$ replaces all occurrences of db in $\rho(v)$ by s. The second rule above applies when the state is a collection type while the third rule applies when the state is a record. The second rule uses the condition $w = v$ to force the comprehension, which reconstructs the collection value, to replace the element v (bound before the update) by the new value e.

For example, at the point of the update c.hotels += ... in the previous example, the binding list ρ binds c into db.cities. In that case, $\mathcal{P}[\![\text{c.hotels}]\!]\,\rho\,e\,s$ is equal to:

$$\begin{array}{l}
\mathcal{P}[\![\text{db.cities}]\!]\,\rho\,(\{\!\{\,\mathcal{P}[\![w.\text{hotels}]\!]\,\rho\,e\,w \mid w \leftarrow \text{s.cities},\ w = c\,\}\!\})\,s \\
= \ \langle \text{cities} = \{\!\{\,\langle \text{hotels} = e, \text{hotel\_num} = w.\text{hotel\_num}\rangle \mid w \leftarrow s.\text{cities},\ w = c\,\}\!\}\rangle
\end{array}$$

The predicate $w = c$ guarantees that only the hotels of the city c (i.e. Arlington) are changed.

One implementation of $\mathcal{P}[\![path]\!]\,\rho\,v\,s$ is $path := v$, which destructively modifies the part of the database reached by *path* into v. But we can do better than that. As it was explained in the introduction, our motivation for using denotational semantics is not to simply give a formal meaning to the destructive constructs but to use the semantics as the actual translation of these constructs and, more importantly, to use this translation in query optimization. To do so, we need to define the inverse function, $\mathcal{I}[\![s]\!]$, of $\mathcal{P}[\![path]\!]$. The $\mathcal{P}[\![path]\!]$ function is needed to translate updates into programs so that they can be optimized, while the inverse function, $\mathcal{I}[\![s]\!]$, is needed to generate destructive updates after optimization. Given a reconstruction of a state s, say s', which is a copy of s except of a number of places where new values v_i are used instead, $\mathcal{I}[\![s]\!]\,s'$ generates a list of destructive updates of the form $path_i := v_i$ such that the composition of all $\mathcal{P}[\![path_i]\!]\,\rho\,v_i\,s$ constructs the state s'. The function $\mathcal{I}[\![e]\!]$ is defined as follows: ($+\!\!+$ is list concatenation)

$$\begin{array}{rcl}
\mathcal{I}[\![\oplus\{\!\{\,e \mid w \leftarrow path, \ldots\,\}\!\}]\!]\,s & = & \mathcal{I}[\![e]\!]\,w \\
\mathcal{I}[\![\langle A_1 = e_1, \ldots, A_n = e_n\rangle]\!]\,s & = & (\mathcal{I}[\![e_1]\!]\,(s.A_1)) +\!\!+ \cdots +\!\!+ \mathcal{I}([\![e_n]\!]\,(s.A_n)) \\
\mathcal{I}[\![s]\!]\,s & = & [\,] \\
\mathcal{I}[\![e]\!]\,s & = & [s := e]
\end{array}$$

For example, $\mathcal{I}[\![\mathcal{P}[\![\text{c.hotels}]\!]\,\rho\,e\,s]\!]\,s$ returns [c.hotels $:=$ e], which is the original update. Under this approach, first, the semantics of a program is given in terms of state transformers and the $\mathcal{P}[\![path]\!]$ state reconstructions are expanded; then, normalization and query optimization take place, which eliminate unnecessary updates; finally, the reconstructed state is transformed into a number of destructive updates to the database using $\mathcal{I}[\![e]\!]$. We leave it for a future work to demonstrate that this framework is as effective as the framework for handling object updates.

Our optimization framework for destructive updates can also be used to handle the view maintenance problem [GM95, CKL97]. In its general form, a view is a function f from the database state db to some value domain. A materialized view, $v \leftarrow f(db)$, stores the view f into the database component, v. Recognizing cases where v can be used directly instead of computing the possibly expensive view f in a query becomes easier after the query is normalized and unnested (since unnesting flattens a query). When the database is updated, the new database becomes $u(db)$, where u is the functional interpretation of the update. Thus, the view maintenance problem is equivalent to expressing $f(u(db))$ in terms of the materialized view v (this is the view recognition problem mentioned above), and transforming all the update primitives in $f(u(db))$ to apply over v instead of db (easily attainable by expressing our state transformations in terms of the update primitives and normalizing the resulting forms). That way, $f(u(db))$ will compute the new materialized view in terms of the old one, v. If we apply the same techniques we used for database updates, the new materialized view $f(u(db))$ can be generated efficiently using destructive updates to v. We are planning to show in a future research that this framework not only requires minimal extensions to our basic framework, but is practical and effective as well.

9 Conclusion

We have presented a formal framework for handling object updates during OODB query optimization. Even though this framework was applied to the monoid comprehension calculus, it can be adapted to work with any optimization framework because many types of object manipulation constructs can be expressed in terms of the basic language constructs by using denotational semantics. Consequently, query optimization applicable to the basic language constructs can be used with minimal changes to remove inefficiencies due to the compositional way of translating programs in denotational semantics. If, in addition, we implement the object store primitives using side effects, the resulting programs can be evaluated as efficiently as programs written by hand.

Acknowledgements: The author is grateful to David Maier for helpful comments on the paper. This work is supported in part by the National Science Foundation under grants IRI-9509955 and IIS-9811525.

References

[AG89] R. Agrawal and N. Gehani. Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++. *Second International Workshop on Database Programming Languages, Salishan, Oregon*, pages 25–40. Morgan Kaufmann Publishers, Inc., June 1989.

[BLS$^+$94] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[Cat94] R. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1994.

[CD92] S. Cluet and C. Delobel. A General Framework for the Optimization of Object-Oriented Queries. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Diego, California*, pages 383–392, June 1992.

[CM95a] S. Cluet and G. Moerkotte. Efficient Evaluation of Aggregates on Bulk Types. Technical report, Aachen University of Technology, October 1995. Technical Report 95-05.

[CM95b] S. Cluet and G. Moerkotte. Nested Queries in Object Bases. In *Fifth International Workshop on Database Programming Languages, Gubbio, Italy*, September 1995.

[Col89] L. S. Colby. A Recursive Algebra and Query Optimization for Nested Relations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, Oregon*, pages 273–283, 1989.

[CKL97] L. Colby, A. Kawaguchi, D. Lieuwen, I. Mumick, and K. A. Ross. Supporting Multiple View Maintenance Policies. In *Proceedings of the ACM SIGMOD Conference on Management of Data, Tucson, Arizona*, pages 405–416, May 1997.

[DGK$^+$91] S. Daniels, G. Graefe, T. Keller, D. Maier, D. Schmidt, and B. Vance. Query Optimization in Revelation, an Overview. *IEEE Data Eng. Bull.*, 14(2):58, June 1991.

[Feg] L. Fegaras. A Uniform Calculus for Collection Types. Oregon Graduate Institute Technical Report 94-030. Available by anonymous ftp from `cse.ogi.edu:/pub/crml/tapos.ps.Z`.

[Feg97] L. Fegaras. An Experimental Optimizer for OQL. University of Texas at Arlington Technical Report TR-CSE-97-007. Available at `http://www-cse.uta.edu/~fegaras/oqlopt.ps.gz`, May 1997.

[Feg98] L. Fegaras. Query Unnesting in Object-Oriented Databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Seattle, Washington*, pages 49–60, June 1998.

[FM95a] L. Fegaras and D. Maier. An Algebraic Framework for Physical OODB Design. In *Fifth International Workshop on Database Programming Languages, Gubbio, Italy*, September 1995.

[FM95b] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. *Proceedings of the ACM SIGMOD International Conference on Management of Data, San Jose, California*, pages 47–58, May 1995.

[FM98] L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. Submitted to TODS. Available at `http://www-cse.uta.edu/~fegaras/monoid.ps.gz`. August 1998.

[GM95] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. Data Engineering Bulletin 18(2): 3-18 (1995).

[KM90] A. Kemper and G. Moerkotte. Advanced Query Processing in Object Bases Using Access Support Relations. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane, Australia*, pages 290–301. Morgan Kaufmann Publishers, Inc., August 1990.

[LD92] D. Lieuwen and D. DeWitt. A Transformation-Based approach to Optimizing Loops in Database Programming Languages. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Diego, California*, pages 91–100, June 1992.

[Oho90] A. Ohori. Representing Object Identity in a Pure Functional Language. In *International Conference on Database Theory, Paris, France*, pages 41–55. Springer-Verlag, December 1990. LNCS 470.

[OW92] Z. Ozsoyoglu and J. Wang. A Keying Method for a Nested Relational Database Management System. In *Proc. IEEE CS Intl. Conf. No. 8 on Data Engineering, Tempe, AZ*, February 1992.

[Pau91] L. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, Inc., 1987.

[Rea89] C. Reade. *Elements of Functional Programming.* Addison Wesley, 1989.

[Sch85] D. Schmidt. Detecting Global Variables in Denotational Specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[SS95] D. Sutton and C. Small. Extending Functional Database Languages to Update Completeness. *BNCOD13 - Thirteenth British National Conference on Databases*, 1995.

[Wad90] Philip Wadler. Comprehending Monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France*, pages 61–78, June 1990.

[Wad92] P. Wadler. The Essence of Functional Programming. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, NM*, pages 1–14, January 1992.

Theorem 1 *The state transformation monoid $\boxed{\oplus}$ is a valid monoid and is compatible with $\oplus$.*

Proof: First we will prove that $Z_{\boxed{\oplus}}$ is the left zero of $\boxed{\oplus}$. The following two proofs are for a non-commutative, non-idempotent monoid. Similar proofs exist for commutative and/or idempotent monoids.

$$\begin{array}{rcl}(Z_{\boxed{\oplus}} \boxed{\oplus} \phi_2)\, s & = & \{\, (v_1 \oplus v_2, s_2) \,|\, (v_1, s_1) \leftarrow \{(Z_\oplus, s)\},\ (v_2, s_2) \leftarrow \phi_2(s_1) \,\} \\ & = & \{\, (Z_\oplus \oplus v_2, s_2) \,|\, (v_2, s_2) \leftarrow \phi_2(s) \,\} \\ & = & \{\, (v_2, s_2) \,|\, (v_2, s_2) \leftarrow \phi_2(s) \,\} \\ & = & \phi_2(s)\end{array}$$

The associativity of $\boxed{\oplus}$ can be proved as follows:

$$\begin{array}{l}((\phi_1 \boxed{\oplus} \phi_2) \boxed{\oplus} \phi_3)\, s \\ = \{\, (v_1 \oplus v_2, s_2) \,|\, (v_1, s_1) \leftarrow \{\, (v_3 \oplus v_4, s_4) \,|\, (v_3, s_3) \leftarrow \phi_1(s),\ (v_4, s_4) \leftarrow \phi_2(s_3) \,\}, \\ \qquad (v_2, s_2) \leftarrow \phi_3(s_1) \,\} \\ = \{\, (v_1 \oplus v_2, s_2) \,|\, (v_3, s_3) \leftarrow \phi_1(s),\ (v_4, s_4) \leftarrow \phi_2(s_3),\ (v_1, s_1) \equiv (v_3 \oplus v_4, s_4), \\ \qquad (v_2, s_2) \leftarrow \phi_3(s_1) \,\} \\ = \{\, ((v_3 \oplus v_4) \oplus v_2, s_2) \,|\, (v_3, s_3) \leftarrow \phi_1(s),\ (v_4, s_4) \leftarrow \phi_2(s_3),\ (v_2, s_2) \leftarrow \phi_3(s_4) \,\} \\ = \{\, (v_3 \oplus (v_4 \oplus v_2), s_2) \,|\, (v_3, s_3) \leftarrow \phi_1(s),\ (v_4, s_4) \leftarrow \phi_2(s_3),\ (v_2, s_2) \leftarrow \phi_3(s_4) \,\} \\ = (\phi_1 \boxed{\oplus} (\phi_2 \boxed{\oplus} \phi_3))\, s\end{array}$$

If $\oplus$ is idempotent, then $\boxed{\oplus}$ is too, since for $\phi_1 = \phi_2 = \phi$ we have $\mathcal{F}(\phi) = \{(Z_\oplus, s_1)\}$ in Definition 2, thus $(\phi \boxed{\oplus} \phi)\, s = \phi(s))$. If T is commutative, then we have $(\phi_1 \boxed{\oplus} \phi_2)\, s \ = \ (\phi_2 \boxed{\oplus} \phi_1)\, s$, since $\mathcal{G}$ in Definition 2 is equal to the set comprehension of $\phi_1 \boxed{\oplus} \phi_2$ if we exchange ϕ_1 with ϕ_2. □

Theorem 2 *If e is an expression of the monoid calculus (without object extensions) that has a monotype, then*

$$e \overset{*}{\rightarrow} a \qquad \Rightarrow \qquad [\![e]\!]\, s \ \overset{*}{\rightarrow} \ \{(a, s)\}$$

where $\rightarrow$ is one-step normalization and $\overset{*}{\rightarrow}$ is normalization in many steps.

Proof: (By structural induction.) We will prove this theorem for Rules (S3) and (S14) only. The proof is similar for the other rules. In Rule (S3), e is $e_1 = e_2$. We assume that the theorem is true for e_1 and e_2 (induction hypothesis). That is, if $e_i \overset{*}{\rightarrow} a_i$, then $[\![e_i]\!]\, s \ \overset{*}{\rightarrow} \ \{(a_i, s)\}$. Then

$$\begin{array}{rcl}[\![e_1 = e_2]\!]\, s & = & \{\, (v_1 = v_2, s_2) \,|\, (v_1, s_1) \leftarrow [\![e_1]\!]\, s,\ (v_2, s_2) \leftarrow [\![e_2]\!]\, s_1 \,\} \\ & \overset{*}{\rightarrow} & \{\, (v_1 = v_2, s_2) \,|\, (v_1, s_1) \leftarrow \{(a_1, s)\},\ (v_2, s_2) \leftarrow \{(a_2, s_1)\} \,\} \\ & \overset{*}{\rightarrow} & \{(a_1 = a_2, s)\} \\ & \overset{*}{\rightarrow} & \{(a, s)\}\end{array}$$

where $(a_1 = a_2) \overset{*}{\rightarrow} a$, that is, $(e_1 = e_2) \overset{*}{\rightarrow} a$ (because of the Church-Rosser property of the normalization algorithm).

For Rule (S14) we assume that the theorem is true for u and $\oplus\{\!\!\{\, e \,|\, \overline{r} \,\}\!\!\}$, i.e., if $u \overset{*}{\to} a$ and $\oplus\{\!\!\{\, e \,|\, \overline{r} \,\}\!\!\} \overset{*}{\to} b$, then $[\![u]\!]\, s \;\overset{*}{\to}\; \{(a,s)\}$ and $[\![\oplus\{\!\!\{\, e \,|\, \overline{r} \,\}\!\!\}]\!]\, s \;\overset{*}{\to}\; \{(b,s)\}$. Then:

$$\begin{array}{ll}
\multicolumn{2}{l}{[\![\oplus\{\!\!\{\, e \,|\, v \leftarrow u,\, \overline{r} \,\}\!\!\}]\!]\, s} \\
\quad = & \{\!\!\{\, (v_2,s_2) \,|\, (v_1,s_1) \leftarrow [\![u]\!]\, s,\, (v_2,s_2) \leftarrow \boxed{\oplus}\{\!\!\{\, [\![\oplus\{\!\!\{\, e \,|\, \overline{r} \,\}\!\!\}]\!] \,|\, v \leftarrow v_1 \,\}\!\!\}\, s_1 \,\}\!\!\} \\
\quad \overset{*}{\to} & \{\!\!\{\, (v_2,s_2) \,|\, (v_1,s_1) \leftarrow \{(a,s)\},\, (v_2,s_2) \leftarrow \boxed{\oplus}\{\!\!\{\, \lambda s.\{(b,s)\} \,|\, v \leftarrow v_1 \,\}\!\!\}\, s_1 \,\}\!\!\} \\
\quad = & \{\!\!\{\, (v_2,s_2) \,|\, (v_1,s_1) \leftarrow \{(a,s)\},\, (v_2,s_2) \leftarrow \{(\oplus\{\!\!\{\, x \,|\, v \leftarrow v_1,\, x \leftarrow b \,\}\!\!\}, s_1)\} \,\}\!\!\} \\
\quad \overset{*}{\to} & \{\!\!\{\, (v_2,s_2) \,|\, (v_2,s_2) \leftarrow \{(\oplus\{\!\!\{\, x \,|\, v \leftarrow a,\, x \leftarrow b \,\}\!\!\}, s)\} \,\}\!\!\} \\
\quad \overset{*}{\to} & \{(\oplus\{\!\!\{\, x \,|\, v \leftarrow a,\, x \leftarrow b \,\}\!\!\}, s)\}
\end{array}$$

But $\oplus\{\!\!\{\, x \,|\, v \leftarrow u,\, x \leftarrow \oplus\{\!\!\{\, e \,|\, \overline{r} \,\}\!\!\} \,\}\!\!\} \overset{*}{\to} \oplus\{\!\!\{\, e \,|\, v \leftarrow u,\, \overline{r} \,\}\!\!\}$ and $\oplus\{\!\!\{\, x \,|\, v \leftarrow u,\, x \leftarrow \oplus\{\!\!\{\, e \,|\, \overline{r} \,\}\!\!\} \,\}\!\!\} \overset{*}{\to} \oplus\{\!\!\{\, x \,|\, v \leftarrow a,\, x \leftarrow b \,\}\!\!\}$, which proves the theorem for this term since according the the Church-Rosser property $\oplus\{\!\!\{\, e \,|\, v \leftarrow u,\, \overline{r} \,\}\!\!\}$ and $\oplus\{\!\!\{\, x \,|\, v \leftarrow a,\, x \leftarrow b \,\}\!\!\}$ have the same canonical form. □