Updating Complex Value Databases *

H. Liefke and **S.B. Davidson**
Dept. of Computer and Information Science
University of Pennsylvania
liefke@seas.upenn.edu and susan@cis.upenn.edu

February 23, 1998

Abstract

Query languages and their optimizations have been a very important issue in the database community. Languages for updating databases, however, have not been studied to the same extent, although they are clearly important since databases must change over time. The structure and expressiveness of updates is largely dependent on the data model. In relational databases, for example, the update language typically allows the user to specify changes to individual fields of a subset of a relation that meets some selection criterion. The syntax is terse, specifying only the pieces of the database that are to be altered. Because of its simplicity, most of the optimizations take place in the internal processing of the update rather than at the language level. In complex value databases, the need for a terse and optimizable update language is much greater, due to the deeply nested structures involved.

Starting with a query language for complex value databases called the Collection Programming Language (CPL), we describe an extension called CPL+ which provides a convenient and intuitive specification of updates on complex values. CPL is a functional language, with powerful optimizations achieved through rewrite rules. Additional rewrite rules are derived for CPL+ and a notion of "deltafication" is introduced to transform complete updates, expressed as conventional CPL expressions, into equivalent update expressions in CPL+. As a result of applying these transformations, the performance of complex updates can increase substantially.

1 Introduction

Although query languages for complex value databases have been well studied [3, 35, 6, 18], the issue of updating such databases has not. For example, a standard for object-oriented query languages is given in [11] (OQL), but there is no mention of updates apart from the notion of a transaction. Update languages – for any model – are, however, clearly important since databases are not static but change to reflect the world that they model. Such changes can trigger updates at the instance as well as the schema level, and while both are important for now we only address changes at the instance level. Schema updates and schema evolution have been studied thoroughly, for example in [27, 28, 16].

What should an update language provide? We believe that users should be able to specify updates by simply describing the parts of the database instance that have to be changed. Update languages for the relational data model, for example, typically allow the user to specify changes to individual fields of a subset of a relation that meets some selection criterion. As an example, suppose our database contains a relation

*This research was supported in part by DOE DE-FG02-94-ER-61923 Sub 1, NSF BIR94-02292 PRIME, ARO AASERT DAAH04-93-G0129, and ARPA N00014-94-1-1086.

Employees(Id, Name, Salary) and we wish to increase the salary of any employee named "Joe" by $5,000. In SQL-92 this could be written as

```
UPDATE Employees E
SET S.Salary= S.Salary+5000
WHERE S.Name= 'Joe'
```

The syntax is terse, specifying only the pieces of the database that are to be altered. In complex value databases, the need for a terse specification of updates is even greater than in relational databases due to the deeply nested structures involved.

The update language should also be optimizable. In relational databases, this is usually interpreted as determining the best way of implementing a selection criterion. Interestingly however, it may be possible for users to write expressions that appear to update a larger subvalue than actually necessary. Suppose that the user wrote the update expression

```
UPDATE Employees E
SET S.Id= S.Id, S.Name='Joe', S.Salary= S.Salary+5000
WHERE S.Name= 'Joe'
```

Clearly this is equivalent to the previous SQL-92 update expression. Although it is unlikely that the second expression will take much of a performance hit with a relational instance since the attribute values are base types (integer, string, boolean, etc), it could be extremely inefficient if attributes are allowed to have complex types such as sets. Thus while there are not many optimizations possible for relational update languages - largely due to the simple data model – this is not true anymore for complex value update languages. Updates (and necessary queries within the update) may involve large nested complex objects, and optimization techniques should reduce the number of values in the database that are read or updated.

Starting with a query language for complex value databases called the Collection Programming Language (CPL), we describe in this paper an extension for updates called CPL+ that provides a terse, optimizable specification for complex updates. Since CPL is a functional language and updates cause side-effects, updates cannot be embedded in CPL expressions. We therefore extend CPL at the top level with constructs for updates. Powerful rewrite rules are derived for CPL+ and a notion of "deltafication" is introduced to generate more efficient CPL+ updates from complete updates. Complete updates completely replace a value in the database by the result value of a query, whereas CPL+ updates replace the parts of a value that actually change. As a result of these transformations, many complex updates can be simplified and performed more efficiently on the database.

The rest of this paper is organized as follows: Section 2 gives an overview of the complex data model, the CPL language, and the internal representation which is used for rewriting and optimizing queries. The new update language CPL+ is introduced in section 3. Section 4 describes the applicable optimizations of CPL+. Rewriting rules for update expressions are defined and the principle of deltafication are explained. We conclude in section 5 with a discussion of related issues on updates and an indication of future research.

2 CPL: A Query Language for Complex Types

The Collection Programming Language (CPL) is based on a complex type system that allows arbitrary nesting of the collection types – set, bag and list – together with record and variant types [10, 35]. For the purposes of this paper we will restrict our attention to sets as the only collection type; we also do not consider issues of object identity, although extensions to CPL can be made along these lines [14]. The set of CPL types $\mathcal{T}$ are therefore given by the syntax:

τ ::= **bool** | **int** | **real** | **string** | ... | $[a_1 : \tau_1, ..., a_n : \tau_n]$ | $< b_1 : \tau_1, ..., b_n : \tau_n >$ | $\{\tau\}$

The types **bool**, **int**, **real**, **string**, etc. are built-in base types and are denoted as $\underline{b}$. $[a_1 : \tau_1, \ldots, a_n : \tau_n]$ constructs record types from the types $\tau_1, \ldots, \tau_n$; $<b_1 : \tau_1, \ldots, b_n : \tau_n>$ constructs variant types from the types $\tau_1, \ldots, \tau_n$; and $\{\tau\}$ constructs set types from the type τ.

For example, the following CPL type Person could be used to represent a person in a company:

```
[Name:string, Age:int, Salary:real,
 Info:<Empl   :[Projects:{string}],
       Manager:[Secr:string, Project:string],
       Secr   :[Manager:string]>]
```

Each person has a name, an age, and a salary, and is either an employee, a manager or a secretary. An employee is affiliated with a set of projects. A manager has a secretary and is affiliated with one single project. A secretary works for a manager.

Values in CPL are explicitly constructed as follows: $[a_1 : e_1, \ldots, a_n : e_n]$ for records, giving values of the appropriate type for each of the attributes; $<a : e>$ for variants, giving a value of the appropriate type for one of the labels; and $\{e_1 \ldots e_n\}$ for sets. For example, a value conforming to the Person type is:

```
[Name:"Tom",Age:43,Salary:34000,Info:<Empl:{"Proj1","Proj3"}>]
```

We consider the database schema to consist of a type, and an instance of the database to be a value of that type. Note that we are ignoring issues of integrity constraints, and assume nothing about the "correctness" of instances.

Continuing with the previous example, the schema of a database for the company could be the following type Company:

```
[Persons:{[Name:string, Age:int, Salary:real,
           Info:<Empl:[Projects:{string}],
                 Manager:[Secr:string, Project:string],
                 Secr:[Manager:string]>]},
 Projs:{[Name:string,Descr:string]}]
```

Here, in addition to sets of people, information about projects is maintained. A valid instance of this database could be the following value:

```
[Persons:{[Name:"Tom"  ,Age:43,Salary:34000,Info:<Empl:{"Proj1","Proj3"}>],
          [Name:"Julie",Age:28,Salary:23000,Info:<Empl:{"Proj1"}>],
          [Name:"Ellen",Age:32,Salary:56000,Info:<Secr:[Manager:"Peter"]>],
          [Name:"Peter",Age:54,Salary:78000,
              Info:<Manager:[Secr:"Ellen",Project:"Proj3"]>]},
 Projs:{[Name:"Proj1",Descr:"Database"],
        [Name:"Proj3",Descr:"Investment"]}]
```

2.1 Queries in CPL

The syntax of CPL resembles, very roughly, that of relational calculus. However there are important differences that make it possible to deal with the richer variety of types we have mentioned.

The important syntactic unit of CPL is the *comprehension*, which can be used with a variety of collection types. As an example of a set comprehension, the following extracts the name and salary of all people who are older than 40 from our database of type Company:

```
{[Name:p.Name,Sal:p.Salary] | \p <- Persons, p.Age>40 }
```

The variable `p` is successively bound (indicated by the backslash, "`\p`") to each element of Persons; "`<-`" is a set-generator. For each `p`, if `p.Age>40` the value `[Name:p.Name,Sal:p.Salary]` is constructed; all such values are then combined in a set to form the final result. Note we use the literal Persons as an entry point to the database. It represents the attribute Persons of the database with record type Company. This example shows the use of comprehension, projection, and conditions.

Another important concept of CPL is *pattern matching*. Instead of binding an element of a collection to a variable name using the expression `\p`, it is possible to specify complex variable bindings and conditions using patterns. The following example, which returns the age and project description of each manager named "Tom", illustrates the use of patterns:

```
{[ManagerAge:a,ProjDescr:p.Descr]  |
   [Name:"Tom",Age:\a,Info:<Manager:[Project:\proj,...]>,...] <- Persons,
   \p<-Projs, p.Name=proj}
```

Each expression in CPL has a type that can be automatically inferred. The type of the result of the last query, for instance, would be `{[ManagerAge:int, ProjDescr:string]}`. The query also illustrates how to form the join of two sets by using two set-generators.

These examples illustrate only a small part of the expressive power of CPL. A more detailed description of the language, including nesting, flattening and function definition, is given in [10]. An important property of comprehension syntax is that it is derived from a more powerful programming paradigm on collection types, that of *structural recursion* [9, 8]. This more general form of computation on collections allows the expression of aggregate functions such as summation, as well as functions such as transitive closure, that cannot be expressed through comprehensions alone. The advantage of using comprehensions is that they have a well-understood set of transformation rules [35, 33, 32] that generalize many of the known optimizations of relational query languages to work for this richer type system.

In addition to the existing language primitives in CPL, many different operators and functions are also available to increase the functionality of CPL. Arithmetic operations, such as `+` and `*`, logical operations like equality tests as well as set operations are an integral part of the system. For the purposes of this paper, we consider only the fragment of CPL without functions or operations on lists and bags. The scope of this language will become clearer in the next subsection.

2.2 The Monad Algebra $\mathcal{NRC}^+$

Similar to the use of the relational algebra as a mathematical framework for implementing and optimizing SQL, CPL is based on an algebra called $\mathcal{NRC}^+$. Since the optimizations we will present, in particular the notion of deltafication, are based on $\mathcal{NRC}^+$, we will make a brief discursion to describe this more manipulable internal form of CPL.

Expressions of the monadic algebra $\mathcal{NRC}^+$ have the following form:

$$
\begin{aligned}
e ::= {} & \underline{c} \mid v \mid \textbf{true} \mid \textbf{false} \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid [a_1 : e_1, ..., a_n : e_n] \mid \pi_a(e) \mid \\
& < b : e > \mid \textbf{case } e \textbf{ of } b_1(v_1) \Rightarrow e_1 , ..., b_n(v_n) \Rightarrow e_n \mid \{\} \mid \{e\} \mid e_1 \cup e_2 \mid \bigcup \{e_1 \mid v \in e_2\}
\end{aligned}
$$

The expression $\underline{c}$ denotes literals like "Tom", 34.5, etc. **true**, and **false** denote the literals for the boolean type. The expression v represents the value of the variable. **if** e_1 **then** e_2 **else** e_3 evaluates to e_2 or e_3, depending on the evaluation of condition e_1. $< b : e >$ is used to construct a variant value that has branch b and branch value e. **case** e **of** $b_1(v_1) \Rightarrow e_1\ , ..., b_n(v_n) \Rightarrow e_n$ deconstructs the variant value e: If e has branch b_i for some $1 \leq i \leq n$, then its branch value is bound to variable v_1 and e_1 is evaluated. Empty sets and singleton sets can be constructed using $\{\}$ and $\{e\}$, respectively. The construct $e_1 \cup e_2$ denotes the union of sets.

The construct that differs most from CPL is $\bigcup\{e_1 \mid v \in e_2\}$: It is the union of the sets $e_1[v \setminus e'_1]$, ..., $e_1[v \setminus e'_n]$ with $e_2 = \{e'_1, ..., e'_n\}$, and captures the implementation of comprehensions in CPL.

The semantics of $\mathcal{NRC}^+$ is described in figure 1 and the typing rules are shown in figure 2.

$$
\begin{array}{rcl}
\mathcal{N}[\![\underline{c}]\!]\rho & \equiv & \underline{c} \\
\mathcal{N}[\![v]\!]\rho & \equiv & \rho(v) \\
\mathcal{N}[\![\mathbf{true}]\!]\rho & \equiv & \mathbf{true} \\
\mathcal{N}[\![\mathbf{false}]\!]\rho & \equiv & \mathbf{false} \\
\mathcal{N}[\![e_1\ \wedge\ e_2]\!]\rho & \equiv & \mathcal{N}[\![e_1]\!]\rho \wedge \mathcal{N}[\![e_2]\!]\rho \\
\mathcal{N}[\![e_1\ \vee\ e_2]\!]\rho & \equiv & \mathcal{N}[\![e_1]\!]\rho \vee \mathcal{N}[\![e_2]\!]\rho \\
\mathcal{N}[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]\rho & \equiv & \left\{ \begin{array}{lcl} \mathcal{N}[\![e_1]\!]\rho = \mathbf{true} & \Rightarrow & \mathcal{N}[\![e_2]\!]\rho \\ \mathcal{N}[\![e_1]\!]\rho = \mathbf{false} & \Rightarrow & \mathcal{N}[\![e_3]\!]\rho \end{array} \right. \\
\mathcal{N}[\![[a_1 : e_1, ..., a_n : e_n]]\!]\rho & \equiv & \{a_i \mapsto \mathcal{N}[\![e_i]\!]\rho \mid 1 \leq i \leq n\} \\
\mathcal{N}[\![\pi_a(e)]\!]\rho & \equiv & (\mathcal{N}[\![e]\!]\rho)(a) \\
\mathcal{N}[\![< b : e >]\!]\rho & \equiv & (b, \mathcal{N}[\![e]\!]\rho) \\
\mathcal{N}[\![\mathbf{case}\ e\ \mathbf{of}\ b_1(v_1) \Rightarrow e_1\ , ..., b_n(v_n) \Rightarrow e_n]\!]\rho & \equiv & \left\{ \begin{array}{lcl} \mathcal{N}[\![e]\!]\rho = (b_1, u) & \Rightarrow & \mathcal{N}[\![e_1]\!]\rho \\ ... & & ... \\ \mathcal{N}[\![e]\!]\rho = (b_n, u) & \Rightarrow & \mathcal{N}[\![e_n]\!]\rho \end{array} \right. \\
\mathcal{N}[\![\{\}]\!]\rho & \equiv & \{\} \\
\mathcal{N}[\![\{e\}]\!]\rho & \equiv & \{\mathcal{N}[\![e]\!]\rho\} \\
\mathcal{N}[\![e_1 \cup e_2]\!]\rho & \equiv & \mathcal{N}[\![e_1]\!]\rho \cup \mathcal{N}[\![e_2]\!]\rho \\
\mathcal{N}[\![\bigcup\{e_1 \mid v \in e_2\}]\!]\rho & \equiv & \bigcup_{x \in \mathcal{N}[\![e_2]\!]\rho} \mathcal{N}[\![e_1]\!]\rho[v \mapsto x]
\end{array}
$$

Figure 1: Denotational Semantics for $\mathcal{NRC}^+$

Let us briefly illustrate the algebra by the following query, which returns the name and salary of people who are older than 40 in our database of type Company.

$$\bigcup\{(\mathbf{if}\ \pi_{Age}(p) > 40\ \mathbf{then}\ \{[Name : \pi_{Name}(p), Salary : \pi_{Salary}(p)]\}\ \mathbf{else}\ \{\}) \mid p \in Persons\}$$

In this query, the variable p is successively bound to each element of Persons. If the age for such a person is bigger than 40, a record consisting of the projected name and salary of the person will be returned in a singleton set. Otherwise, the empty set is returned. The results of these evaluations for all elements of Persons are unioned together into one set. This $\mathcal{NRC}^+$ expression corresponds to the CPL expression in the first example of the previous subsection.

It is not difficult to transform arbitrary CPL expressions into expressions of $\mathcal{NRC}^+$. Perhaps the most interesting is the transformation of comprehensions into a construct $\bigcup\{e_1 \mid v \in e_2\}$; this translation was first

$$\frac{H \vdash e : \tau_1 \quad \tau_1 \leq_{\mathcal{NRC}^+} \tau_2}{H \vdash e : \tau_2} \qquad \frac{v \notin H'}{H, v : \tau, H' \vdash v : \tau}$$

$$\frac{}{H \vdash \mathbf{true} : \mathbf{bool}} \qquad \frac{}{H \vdash \mathbf{false} : \mathbf{bool}}$$

$$\frac{H \vdash e_1 : \mathbf{bool} \quad H \vdash e_2 : \mathbf{bool}}{H \vdash e_1 \ \wedge \ e_2 : \mathbf{bool}} \qquad \frac{H \vdash e_1 : \mathbf{bool} \quad H \vdash e_2 : \mathbf{bool}}{H \vdash e_1 \ \vee \ e_2 : \mathbf{bool}}$$

$$\frac{H \vdash e_1 : \tau_1 \quad \cdots \quad H \vdash e_n : \tau_n}{H \vdash [a_1 : e_1, ..., a_n : e_n] : [a_1 : \tau_1, \cdots, a_n : \tau_n]} \qquad \frac{H \vdash e : [a_1 : \tau_1, \cdots, a_n : \tau_n] \quad 1 \leq i \leq n}{H \vdash \pi_{a_i}(e) : \tau_i}$$

$$\frac{}{H \vdash \{\} : \{\top\}} \qquad \frac{H \vdash e : \tau}{H \vdash \{e\} : \{\tau\}}$$

$$\frac{H \vdash e_1 : \{\tau\} \quad H \vdash e_2 : \{\tau\}}{H \vdash e_1 \cup e_2 : \{\tau\}} \qquad \frac{H, v : \tau_1 \vdash e_1 : \tau_2 \quad H \vdash e_2 : \{\tau_1\}}{H \vdash \bigcup\{e_1 \mid v \in e_2\} : \tau_2}$$

$$\frac{H \vdash e_1 : \mathbf{bool} \quad H \vdash e_2 : \tau \quad H \vdash e_3 : \tau}{H \vdash \mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3 : \tau} \qquad \frac{H \vdash e : \tau}{H \vdash < b : e > : < b : \tau >}$$

$$\frac{H \vdash e : < b_1 : \tau_1 \cdots b_n : \tau_n > \quad H, v_1 : \tau_1 \vdash e_1 : \tau \quad \cdots \quad H, v_n : \tau_n \vdash e_n : \tau}{H \vdash \mathbf{case} \ e \ \mathbf{of} \ b_1(v_1) \Rightarrow e_1 \ , ..., b_n(v_n) \Rightarrow e_n : \tau}$$

Figure 2: Typing rules for $\mathcal{NRC}^+$

studied by Wadler [34] and is based on three important rules:

$$\begin{array}{rcl} \{ \ e \mid \ \} & \Rightarrow & \{e\} \\ \{ \ e_1 \mid \ \backslash v \leftarrow e_2, ...\} & \Rightarrow & \bigcup\{\{e_1 \mid ...\} \mid v \in e_2\} \\ \{ \ e_1 \mid e_2, ...\} & \Rightarrow & \mathbf{if} \ e_2 \ \mathbf{then} \ \{e_1 \mid ...\} \ \mathbf{else} \ \{\} \end{array}$$

Similarly, there exist transformation rules for all types of CPL expressions. The complete set of transformaton rule can be found in [35].

As another example, consider a query that returns the name and project of all managers who have `"Julie"` as a secretary. In CPL, the information could be extracted in the following way:

```
{[Name:n,Project:p] |
    [Name:\n,Info:<Manager:[Secr:"Julie",Project:\p]>] <- Persons}];
```

This query rewrites to the following $\mathcal{NRC}^+$ query:

$$\begin{array}{l} \bigcup\{\mathbf{case} \ \pi_{Info}(p) \ \mathbf{of} \ Empl(e) \Rightarrow \{\}, \\ \qquad\qquad\qquad\qquad\quad Secr(s) \Rightarrow \{\}, \end{array}$$

$$
\begin{array}{ll}
& Manager(m) \Rightarrow (\textbf{if}\ \ \pi_{Secr}(m) = "Julie" \\
& \qquad\qquad\qquad\qquad \textbf{then}\ \{[Name : \pi_{Name}(p), Project : \pi_{Project}(m)]\} \\
& \qquad\qquad\qquad\qquad \textbf{else}\ \{\}) \\
\mid p \in Persons\}
\end{array}
$$

The details of the transformation process and the underlying rules are described in [35]. Furthermore, based on the mathematical properties of monads, a number of important rewriting rules (such as vertical loop fusion and filter promotion) have been developed that generalize to this richer type system many of the well-known optimizations for the relational algebra.

3 Extending CPL with Updates : CPL+

An update is a function from an instance of a given database schema (complex type) to another instance of that schema. For example, if our database schema were `{int}` and initial instance were `{3}`, an update could yield a final instance of `{3,4}` but not `"Tom"`.

This example is a little unusual, of course, as we tend to think of a database schema as giving a set of *named* values, as in the example from the previous sections. Here, the schema has two named values at the top level – Persons and Projects – and the database is a record type. While it may be convenient to think of a database schema as a set of named values, the update language we will describe will work for schemas of any complex type. Top level updates must merely be correct with respect to the database type given.

It is perfectly possible in CPL to specify an update by expressing it completely. Using the Company schema of the previous section, if we want to increase Tom's salary by \$5000 we could write something like the following[1]:

```
@Persons:={[Name:n,Age:a,Info:i,Salary:
             if (n="Tom") then s+5000 else s)]
            | [Name:\n,Age:\a,Salary:\s,Info:\i]<-Persons};
```

However, this form of complete update is cumbersome to specify and inefficient if executed as written – we are rewriting the entire set of Persons when only one record of one set is updated. We will therefore extend CPL with constructs for partially updating values.

Since updates have side-effects, we cannot consider updates as CPL expressions that can be used in other CPL expressions. Instead, it is necessary to introduce a new construct at the root level of the language in order to distinguish between queries and updates:

$$command ::= expr \ \mid\ @upd$$

The prefix @ is used to distinguish between conventional CPL expressions *expr* and the newly introduced CPL+ expressions. The update expression *upd* is given by the following grammar:

$$
\begin{array}{ll}
upd ::= & ID_{upd} \ \mid \\
& a_i := upd \ \mid \\
& b : upd \ \mid \\
& \textbf{if}\ c\ \textbf{then}\ upd_1\ \textbf{else}\ upd_2 \ \mid \\
& \textbf{if}\ c\ \textbf{then}\ upd \ \mid \\
& e \ \mid
\end{array}
$$

[1] Person would actually need to be defined as a parameterless function, and updated using the syntax let Person == ...., but this merely complicates the point.

$\{\ su\ \}$ |
setins e |
$upd_1\ ;\ upd_2$ |
$pat \Rightarrow upd$ |
let $v := e$ **in** upd |

Here, e denotes a conventional CPL expression, and c is a CPL-expression of type **bool**. The construct *pat* denotes a *pattern matching* expression, and *su* is used to express *set updates*. The syntax and meaning of patterns and set updates is described later.

The *identity update* ID_{upd} represents the identity function, and does not change the database instance. The *record update* $a_i := upd$ updates the attribute a_i for the given record value by performing update *upd* on it. The *variant update* $b : upd$ tests if the branch for the given variant value is b; if it is, then the update *upd* is performed on the branch value. Expression **if** c **then** upd_1 **else** upd_2 evaluates condition c and, depending on the result, performs either upd_1 or upd_2; **if** c **then** *upd* is syntactic sugar for **if** c **then** *upd* **else** ID_{upd}.

The update expression e denotes a *complete update*: The value is completely replaced by the value of e. The *collection update* $\{su\}$ updates each element of the set by su. The expression **setins** e updates a set by inserting the elements of set e. $upd_1\ ;\ upd_2$ denotes a *sequence* of updates. The expression **let** $v := e$ **in** *upd* binds the evaluation of e to v and performs update *upd*.

As in CPL, we use patterns to select subvalues and introduce variables. The update expression $pat \Rightarrow upd$ tries to match the value that has to be updated with the pattern. If the value matches, the variables in *pat* are bound to the respective parts in the value and the update *upd* is performed. If the value does not match the pattern, then the value will not be changed.

The syntax of patterns is:

$$pat ::= \_ \mid e \mid \backslash v \mid\ < b : pat >\ \mid [a_1 : pat_1, \cdots, a_n : pat_n]$$

The pattern _ matches anything. Pattern e matches the object whose value is equal to the value of e. The pattern $\backslash v$ binds variable name v to the value. The pattern $< b : pat >$ matches those variant values that have branch b. The pattern *pat* is matched with the respective branch value. Pattern $[a_1 : pat_1, \cdots, a_n : pat_n]$ matches record values that have attributes a_1, ..., a_n and whose attribute values match the patterns pat_1, ..., pat_n; the record value can have other attributes in addition to those in the pattern, i.e. it matches records *partially*. Whenever a pattern *pat* in $pat \Rightarrow upd$ is not matched, then the update *upd* will not be performed on the value in the database. Otherwise, the update *upd* is performed while the new variables from *pat* can be used in *upd*.

It is important to note that the variables in a pattern are bound to the *old* value. That means the update *upd* does not have any impact on the value represented by the variables. For example, let us consider the update `\a => (a+2;a+5)`, which can be applied to any integer value. The first update `a+2` increases the value by 2; the second update `a+5` increases the value by 5. However, the variable `a` in the second update represents the value *before* the first update occured. Therefore, the entire update will only increase the value by 5. To increase the value by 7, it is necessary to bind the variable in each of the updates: `(\a => a+2;\a => a+5)`.

Set updates transform set values into new set values, in which elements may be inserted, deleted or modified. The expressions below are applied to each of the elements in the set:

su ::= **setdel** |
upd |
setrepl e |
setif c **then** su_1 **else** su_2 |

$\textbf{setif}\ c\ \textbf{then}\ su\ \ |$
$su_1\ ;\ su_2\ \ |$
$pat \Rightarrow su\ \ |$
$\textbf{setlet}\ v := e\ \textbf{in}\ su\ \ |$

The primitive **setdel** deletes the element from the set. The set update *upd* applies *upd* to the element of the set. Expression **setrepl** e evaluates the set expression e for the element and includes elements of the resulting set in the overall set. Expression **setif** c **then** su_1 **else** su_2 evaluates condition c and depending on the result, performs either su_1 or su_2 on the element of the set. The construct **setif** c **then** su is syntactic sugar and can be expanded to **setif** c **then** su **else** ID_{upd}. The sequence expression $su_1\ ;\ su_2$ performs su_1 and su_2 sequentially on the set element. The update expression $pat \Rightarrow su$ tries to match the element of the set against the pattern. If the value matches, then the update su is performed on the value. Otherwise, the value will not be changed. The expression **setlet** $v := e$ **in** su binds the evaluation of e to v and performs set update su on the set element.

Let us consider a few update examples for the Company database:

Example 1: Increase Tom's salary by $5000.

```
@Persons:={ \e => if e.Name="Tom" then (Salary:=e.Salary+5000)}
```

Variable `e` is successively bound to each element of the set Persons. If the condition `e.Name="Tom"` evaluates to true for an element, then the salary is raised by $5000. Note that it is also possible to use pattern matching to express the condition over the name and to bind a variable to salary:

```
@Persons:={ [Name:"Tom",Salary:\s] => (Salary:=s+5000)}
```

These should be contrasted with the complete update expession given at the beginning of this section. Rather than specifying the complete new set of persons, only the part that has to be updated is mentioned.

Example 2: Insert a new manager named Jim with salary=$45000, age=64, a secretary named Ellen, and a project "NewProj". Ellen has age=25 and salary=$23000; "NewProj" is a new project in the database.

```
@(Persons:=setins {
     [Name:"Jim",Age:64,Salary:45000,
      Info:<Manager:("Ellen","NewProj")>],
     [Name:"Ellen",Age:25,Salary:23000,Info:<Secretary:[Manager:"Jim"]>]};
  Projs:=setins {[Name:"NewProj", Descr:"Technology"]})
```

In the example, the primitive **setins** is used to insert two new elements into the set Persons and one new element into the set Projs.

Example 3: Delete project "Test" from the database. This entails deleting it from the project list of all employees, removing all employees who were only affiliated with that project, replacing the project name of the associated managers by "Void", and removing the project "Test" from the list of projects.

First, "Test" is removed from the project set:

```
@Projs:={[Name:"Test"] => setdel}
```

All employees who are only affiliated with "Test" are then removed:

```
@Persons:={[Info:<Empl:[Projects:{"Test"}]>] => setdel}
```

The project is removed from the set of projects for each remaining employee:

```
@Persons:={\e => Info:= Empl: Projects:={"Test" => setdel}}
```

If the project of a manager is "Test", then replace it by "Void":

```
@Persons:={[Info:<Manager:[Project:"Test"]>)
          => (Info:=Manager:[Project:="Void"])}
```

The updates can now easily be combined into one single update on the database:

```
@(Projs:={[Name:"Test"] => setdel};
  Persons:={[Info:<Empl:[Projects:{"Test"}]>] => setdel};
  Persons:={\e => (Info:= Empl: Projects:={"Test" => setdel})};
  Persons:={[Info:<Manager:[Project:"Test"]>]
            => (Info:=Manager: Project:="Void")}
)
```

Note that update expressions can only be applied to values of the correct type. An update of the form `Persons:=...`, for instance, can only be applied to record values that have a Persons field. The update expression in the first example is only correct for record values that minimally contain an attribute Persons which is a set with an element type that minimally contains two attributes Name and Salary with the base types **string** and **real** respectively.

3.1 Semantics of CPL+

An update can be interpreted as a transformation of an old value in the database into a new value: It is possible to express the new value as an $\mathcal{NRC}^+$ expression as figure 3 shows.

For patterns and collection updates, two auxiliary semantic functions $\mathcal{P}[\![pat, pv, upd]\!]_\tau\rho$ and $[\![su]\!]^s_\tau\rho$ are defined. The collection primitives translate a set of values to a new set of values. Therefore, they are interpreted as functions from $[\![\{\tau\}]\!]$ to $[\![\{\tau\}]\!]$. The semantics of all collection updates is defined in figure 4.

Patterns such as in $pat \Rightarrow upd$ introduce variables and impose conditions on the update value. The new variable bindings must be represented as an extension to the variable environment ρ before evaluating upd. Conditions are imposed on parts of the updated value. They are represented using the $\mathcal{NRC}^+$ expressions **if** c **then** upd_1 **else** upd_2 and **case** e **of**....

The semantics of patterns is defined by function $\mathcal{P}[\![pat, pv, upd]\!]_\tau\rho$ shown in figure 5. This function is defined recursively on the structure of pat. It interprets the pattern expression pat with respect to the value in pv. It reduces pattern expressions based on its structure to the semantics of the update upd with additional.

Note the recursive definition for record patterns. Within a record pattern, variables newly defined in an attribute pattern can be used in subsequent attribute patterns. For instance, the pattern $[a : \backslash v, b : v]$ matches record value that have attributes a and b that have the same attribute value.

Like $[\![upd]\!]_\tau\rho$ and $[\![upd]\!]^s_\tau\rho$ the semantic functions $\mathcal{P}[\![pat, pv, upd]\!]_\tau\rho$ and $\mathcal{P}_s[\![pat, pv, upd]\!]_\tau\rho$ are functions from $[\![\tau]\!]$ to $[\![\tau]\!]$ and $[\![\{\tau\}]\!]$ to $[\![\{\tau\}]\!]$, respectively. Figure 5 shows the definition of $\mathcal{P}[\![pat, pv, upd]\!]_\tau\rho$ and $\mathcal{P}_s[\![pat, pv, upd]\!]_\tau\rho$.

$$
\begin{array}{rcl}
\llbracket ID_{upd} \rrbracket_{\tau} \rho(v) & \equiv & v \\
\llbracket a_i := upd \rrbracket_{[a_1:\tau_1,\ldots,a_n:\tau_n]} \rho(v) & \equiv & [a_1 : \pi_{a_1}(v), \cdots, a_{i-1} : \pi_{a_{i-1}}(v), \\
 & & a_i : \llbracket upd \rrbracket_{\tau_i} \rho(\pi_{a_i}(v)), \\
 & & a_{i+1} : \pi_{a_{i+1}}(v), \cdots, a_n : \pi_{a_n}(v)] \\
\llbracket b_i : upd \rrbracket_{<b_1:\tau_1,\ldots,b_n:\tau_n>} \rho(v) & \equiv & \textbf{case } v \textbf{ of } b_1(v_1) \Rightarrow v, \cdots, b_{i-1}(v_{i-1}) \Rightarrow v, \\
 & & b_i(v_i) \Rightarrow < b_i : \llbracket upd \rrbracket_{\tau_i} \rho(v_i) >, \\
 & & b_{i+1}(v_{i+1}) \Rightarrow v, \cdots b_n(v_n) \Rightarrow v \\
\llbracket \textbf{if } c \textbf{ then } upd \rrbracket_{\tau} \rho(v) & \equiv & \llbracket \textbf{if } c \textbf{ then } upd \textbf{ else } ID_{upd} \rrbracket_{\tau} \rho(v) \\
\llbracket \textbf{if } c \textbf{ then } upd_1 \textbf{ else } upd_2 \rrbracket_{\tau} \rho(v) & \equiv & \textbf{if } c \textbf{ then } \llbracket upd_1 \rrbracket_{\tau} \rho(v) \textbf{ else } \llbracket upd_2 \rrbracket_{\tau} \rho(v) \\
\llbracket e \rrbracket_{\tau} \rho(v) & \equiv & e \\
\llbracket \{su\} \rrbracket_{\{\tau\}} \rho(s) & \equiv & \bigcup \{ \llbracket su \rrbracket^s_{\tau} \rho(v) \mid v \in s \} \\
\llbracket \textbf{setins } e \rrbracket_{\{\tau\}} \rho(v) & \equiv & v \cup e \\
\llbracket upd_1 \ ; \ upd_2 \rrbracket_{\tau} \rho(v) & \equiv & (\llbracket upd_1 \rrbracket_{\tau} \rho \circ \llbracket upd_2 \rrbracket_{\tau} \rho)(v) \\
\llbracket pat \Rightarrow upd \rrbracket_{\tau} \rho(v) & \equiv & \mathcal{P} \llbracket pat, v, upd \rrbracket_{\tau} \rho(v) \\
\llbracket \textbf{let } w := e \textbf{ in } upd \rrbracket_{\tau} \rho(v) & \equiv & \llbracket upd \rrbracket_{\tau} \rho[w \mapsto e](v)
\end{array}
$$

Figure 3: Denotational Semantics for CPL+

$$
\begin{array}{rcl}
\llbracket \textbf{setdel} \rrbracket^s_{\tau} \rho(v) & \equiv & \{\} \\
\llbracket upd \rrbracket^s_{\tau} \rho(v) & \equiv & \{ \llbracket upd \rrbracket_{\tau} \rho(v) \} \\
\llbracket \textbf{setrepl } e \rrbracket^s_{\tau} \rho(v) & \equiv & e \\
\llbracket \textbf{setif } c \textbf{ then } su_1 \textbf{ else } su_2 \rrbracket^s_{\tau} \rho(v) & \equiv & \textbf{if } c \textbf{ then } \llbracket su_1 \rrbracket^s_{\tau} \rho(v) \textbf{ else } \llbracket su_2 \rrbracket^s_{\tau} \rho(v) \\
su_1 \ ; \ su_2 \tau \rho(v) & \equiv & \bigcup \{ \llbracket su_2 \rrbracket^s_{\rho}(x) \mid x \in \llbracket su_1 \rrbracket^s_{\rho}(v) \} \\
\llbracket pat \Rightarrow upd \rrbracket^s_{\tau} \rho(v) & \equiv & \mathcal{P}_s \llbracket pat, v, upd \rrbracket_{\tau} \rho(v) \\
\llbracket \textbf{setlet } w := e \textbf{ in } su \rrbracket^s_{\tau} \rho(v) & \equiv & \llbracket upd \rrbracket^s_{\tau} \rho[w \mapsto e](v)
\end{array}
$$

Figure 4: Semantics for set updates in CPL+

$$
\begin{array}{rcl}
\mathcal{P}[\![\_, pv, upd]\!]_\tau \rho & \equiv & [\![upd]\!]_\tau \rho \\
\mathcal{P}[\![\backslash v, pv, upd]\!]_\tau \rho & \equiv & [\![upd]\!]_\tau \rho[v \mapsto pv] \\
\mathcal{P}[\![e, pv, upd]\!]_\tau \rho & \equiv & [\![\textbf{if } e = pv \textbf{ then } upd]\!]_\tau \rho \\
\mathcal{P}[\![< b : p >, pv, upd]\!]_\tau \rho & \equiv & \textbf{case } pv \textbf{ of } b(v') \Rightarrow \mathcal{P}[\![p, v', upd]\!]_\tau \rho \\
\mathcal{P}[\![[a_1 : p_1, \cdots, a_n : p_n], pv, upd]\!]_\tau \rho & \equiv & \mathcal{P}[\![p_1, \pi_{a_1}(pv), [a_2 : p_2, \cdots, a_n : p_n] \Rightarrow upd]\!]_\tau \rho \\
\mathcal{P}[\![[a_1 : p_1], pv, upd]\!]_\tau \rho & \equiv & \mathcal{P}[\![p_1, \pi_{a_1}(pv), upd]\!]_\tau \rho \\
\\
\mathcal{P}_s[\![\_, pv, su]\!]_\tau \rho & \equiv & [\![su]\!]^s_\tau \rho \\
\mathcal{P}_s[\![\backslash v, pv, su]\!]_\tau \rho & \equiv & [\![su]\!]^s_\tau \rho[v \mapsto pv] \\
\mathcal{P}_s[\![e, pv, su]\!]_\tau \rho & \equiv & [\![\textbf{setif } e = pv \textbf{ then } su]\!]^s_\tau \rho \\
\mathcal{P}_s[\![< b : p >, pv, su]\!]_\tau \rho & \equiv & \textbf{case } pv \textbf{ of } b(v') \Rightarrow \mathcal{P}_s[\![p, v', su]\!]_\tau \rho \\
\mathcal{P}_s[\![[a_1 : p_1, \cdots, a_n : p_n], pv, su]\!]_\tau \rho & \equiv & \mathcal{P}_s[\![p_1, \pi_{a_1}(pv), [a_2 : p_2, \cdots, a_n : p_n] \Rightarrow su]\!]_\tau \rho \\
\mathcal{P}_s[\![[a_1 : p_1], pv, su]\!]_\tau \rho & \equiv & \mathcal{P}_s[\![p_1, \pi_{a_1}(pv), su]\!]_\tau \rho
\end{array}
$$

Figure 5: Semantics for patterns in CPL+

3.2 Typing rules

As noted earlier an update expression is applicable to values of different types. This describes the *type* of an update. Let us consider the update $(a := 3; b := c : \{\textbf{setdel}\})$. Obviously, an update value for this update must have two attributes a and b (but it can have more attributes). Attribute a must have type **int** and attribute b must be a variant type that can have branch c which would have to be of a set type. One can observe that this description specifies a *partial type*. A second update example is $a := [e : \text{“}Tom'', f : 5]$ where $[e : \text{“}Tom'', f : 5]$ is a complete record value. Obviously, this update can only be applied to record values that have an attribute a which has the *complete type* $[e : \textbf{string}, f : \textbf{int}]$.

In order to express such complex update types and to make a distinction between partial and complete types, we introduce a type system for updates:

$$
\upsilon ::= \top \mid \bot \mid \overline{\tau} \mid [a_1 : \upsilon_1, \cdots, a_n : \upsilon_n]_u \mid \; < b_1 : \upsilon_1, \cdots, b_n : \upsilon_n >_u \; \mid \{\upsilon\}_u
$$

An update of a certain update type can only be performed on values of certain types. We say that the update *accepts* the value and its type. The update types $\top$ and $\bot$ denote the maximum and minimum type, respectively. Updates of type $\top$ are *invalid* in that they don't accept values of any type. Updates of type $\bot$ accept values of any type. Updates of type $\overline{\tau}$ only accept values of type τ. Updates of type $[a_1 : \upsilon_1, \cdots, a_n : \upsilon_n]_u$ accept record values that have attributes a_1, ..., a_n whose values can be modified by updates of types υ_1, ..., υ_n. An update of type $< b_1 : \upsilon_1, \cdots, b_n : \upsilon_n >_u$ will accept a variant value that, if it has branch b_1, ..., b_n, then its branch value has to be acceptable by the updates of type υ_1, ..., υ_n, respectively. And finally, an update of type $\{\upsilon\}$ accepts a set value whose elements are accepted by updates of type υ.

By this definition, we can easily conclude that update $(a := 3; b := c : \{\textbf{setdel}\})$ has type $[a : \overline{\textbf{int}}, b :< c : \{\bot\}_u >_u]_u$ and update $a := [e : \text{“}Tom'', f : 5]$ has type $[a : \overline{[e : \textbf{string}, f : \textbf{int}]}]_u$, respectively.

An update of a certain (update) type accepts values of certain types. Vice versa, a value of a certain (value) type is accepted by updates of certain types. If updates of type υ_1 accepts all the values (and its types) that an update of type υ_2 accepts, then υ_1 is a *subtype* of υ_2, or $\upsilon_1 \preceq \upsilon_2$. For instance, the update

$a := \backslash v => v + 4$ accepts all the values that are accepted by update ($a := 3; b := c : \{\textbf{setdel}\}$). The subtyping rules are described in figure 6.

$$\frac{}{\upsilon \preceq \top} \qquad \frac{}{\bot \preceq \upsilon}$$

$$\frac{}{\upsilon \preceq \upsilon} \qquad \frac{\upsilon_1 \preceq \upsilon_2}{\{\upsilon_1\}_u \preceq \{\upsilon_2\}_u}$$

$$\frac{\upsilon_1 \preceq \upsilon_1' \quad \cdots \quad \upsilon_m \preceq \upsilon_m' \qquad m \leq n}{[a_1 : \upsilon_1, \cdots, a_m : \upsilon_m]_u \preceq [a_1 : \upsilon_1', \cdots, a_n : \upsilon_n']_u} \qquad \frac{\upsilon_1 \preceq \upsilon_1' \quad \cdots \quad \upsilon_m \preceq \upsilon_m' \qquad m \leq n}{< b_1 : \upsilon_1, \cdots, b_m : \upsilon_m >_u \preceq < b_1 : \upsilon_1', \cdots, b_n : \upsilon_n' >_u}$$

$$\frac{\upsilon_1 \preceq \overline{\tau_1} \quad \cdots \quad \upsilon_n \preceq \overline{\tau_n}}{[a_1 : \upsilon_1, \cdots, a_n : \upsilon_n]_u \preceq \overline{[a_1 : \tau_1, ..., a_n : \tau_n]}} \qquad \frac{\upsilon_1 \preceq \overline{\tau_1} \quad \cdots \quad \upsilon_n \preceq \overline{\tau_n}}{< b_1 : \upsilon_1, \cdots, b_n : \upsilon_n >_u \preceq \overline{< b_1 : \tau_1, ..., b_n : \tau_n >}}$$

$$\frac{\upsilon \preceq \overline{\tau}}{\{\upsilon\}_u \preceq \overline{\{\tau\}}} \qquad \frac{}{\overline{\tau} \preceq \overline{\tau}}$$

Figure 6: Subtyping for Update types

Based on this definition of update types and the associated subtyping relation, the typing rules for update constructs can be obtained. The subsumption rule follows directly from the subtyping: An update of type υ_1 can be considered to be an update of type υ_2 if, and only if, $\upsilon_1 \preceq \upsilon_2$. By definition, the predicate $H \vdash \mathit{upd} : \upsilon$ is true if, and only if, under a type assignment H, it is provable that *upd* has type υ. The specification of typing rules can be found in figure 7.

$$\frac{H \vdash \mathit{upd} : \upsilon_1 \quad \upsilon_1 \preceq \upsilon_2}{H \vdash \mathit{upd} : \upsilon_2} \qquad \frac{}{H \vdash ID_{\mathit{upd}} : \bot} \qquad \frac{H \vdash \mathit{upd} : \upsilon}{H \vdash a := \mathit{upd} : [a : \upsilon]_u}$$

$$\frac{H \vdash \mathit{upd} : \upsilon}{H \vdash b : \mathit{upd} : < b : \upsilon >_u} \qquad \frac{H \vdash e : \tau \quad H, v : \tau \vdash \mathit{upd} : \upsilon}{H \vdash \textbf{let } v := e \textbf{ in } \mathit{upd} : \upsilon} \qquad \frac{H \vdash e : \tau}{H \vdash e : \overline{\tau}}$$

$$\frac{H \vdash_s \mathit{su} : \{\upsilon\}_u}{H \vdash \{\mathit{su}\} : \{\upsilon\}_u} \qquad \frac{H \vdash e : \{\tau\}}{H \vdash \textbf{setins } e : \{\overline{\tau}\}_u} \qquad \frac{H \vdash \mathit{upd}_1 : \upsilon \quad H \vdash \mathit{upd}_2 : \upsilon}{H \vdash \mathit{upd}_1 \ ; \ \mathit{upd}_2 : \upsilon}$$

$$\frac{H \vdash c : \textbf{bool} \quad H \vdash \mathit{upd}_1 : \upsilon \quad H \vdash \mathit{upd}_2 : \upsilon}{H \vdash \textbf{if } c \textbf{ then } \mathit{upd}_1 \textbf{ else } \mathit{upd}_2 : \upsilon} \qquad \frac{H \vdash (\mathit{pat} : \upsilon \rhd G) \quad H, G \vdash \mathit{upd} : \upsilon}{H \vdash \mathit{pat} \Rightarrow \mathit{upd} : \upsilon}$$

Figure 7: Typing rules for CPL+

Set updates such as **setdel** operate on sets of values. We use a slightly different notation for the typing of set updates: $H \vdash_s \mathit{su} : \{\upsilon\}_u$. This voids confusion, since normal update expression can also appear as set updates. The semantics of set updates is shown in figure 8.

Let us finally consider pattern expressions $\mathit{pat} \Rightarrow \mathit{upd}$. A pattern *pat* also has an update types in the sense

$$\frac{}{H \vdash_s \textbf{setdel} : \{\bot\}_u} \qquad \frac{H \vdash upd : \upsilon}{H \vdash_s upd : \{\upsilon\}_u} \qquad \frac{H \vdash e : \tau \quad H, v : \tau \vdash_s su : \{\upsilon\}_u}{H \vdash_s \textbf{setlet}\ v := e\ \textbf{in}\ su : \{\upsilon\}_u}$$

$$\frac{H \vdash_s su_1 : \{\tau\}_u\ H \vdash_s su_2 : \{\tau\}_u}{H \vdash_s su_1\ ;\ su_2 : \{\tau\}_u} \qquad \frac{H \vdash e : \{\tau\}}{H \vdash_s \textbf{setrepl}\ e : \{\upsilon\}_u} \qquad \frac{H \vdash (pat : \upsilon \rhd G)\ H, G \vdash_s su : \{\upsilon\}_u}{H \vdash_s pat \Rightarrow su : \{\upsilon\}_u}$$

$$\frac{H \vdash c : \textbf{bool} \quad H \vdash_s su_1 : \{\tau\}_u \quad H \vdash_s su_2 : \{\tau\}_u}{H \vdash_s \textbf{setif}\ c\ \textbf{then}\ su_1\ \textbf{else}\ su_2 : \{\tau\}_u}$$

Figure 8: Typing rules for set updates in CPL+

that it only "accept" values of certain types. Furthermore, patterns impose type restrictions on the newly introduced variables. these variables might be used in subsequent subexpressions of the pattern or in *upd*, thus having an impact on their types. Thus, they must be included in respective type assignments H. By definition, the sentence $H \vdash (pat : \upsilon \rhd G)$ is true, if, and only if, under a type assignment H, the pattern *pat* has update type υ and the introduced variables have types as described in the generated type assignment G. Type assignment G only includes the new variables and their types.

Based on this definition, the typing rules for pattern expressions can be obtained easily. Figure 9 shows the typing rules for patterns. Note that the type of record patterns is defined recursively on the number of attributes, since newly introduced variables can be used in subsequent attribute patterns.

$$\frac{H \vdash (p : \upsilon_1 \rhd G) \quad \upsilon_1 \preceq \upsilon_2}{H \vdash (p : \upsilon_2 \rhd G)} \qquad \frac{\upsilon \preceq \overline{\tau}}{H \vdash (\backslash v : \upsilon \rhd v : \tau)}$$

$$\frac{H \vdash (p : \upsilon \rhd G)}{H \vdash (< b : p > : < b : \upsilon >_u \rhd G)} \qquad \frac{H \vdash e : \tau}{H \vdash (e : \overline{\tau} \rhd)}$$

$$\frac{H \vdash (p_1 : \upsilon_1 \rhd G_1) \quad H, G_1 \vdash ([a_2 : p_2, \cdots, a_n : p_n] : [a_2 : \upsilon_2, \cdots, a_n : \upsilon_n]_u \rhd G_2)}{H \vdash ([a_1 : p_1, \cdots, a_n : p_n] : [a_1 : \upsilon_1, \cdots, a_n : \upsilon_n]_u \rhd G_1, G_2)}$$

Figure 9: Typing rules for patterns in CPL+

Note that a pattern imposes restrictions about the type of the value and the value itself. A pattern such as $< b : "Test" >$ matches only the value $< b : "Test" >$. However, it is important to distinguish between type correctness and value matching. A value of type $< a : \textbf{bool}, b : \textbf{string} >$ would surely have the correct type for the pattern, but it does not necessarily have to match the pattern. This distinction is necessary since the application of a pattern to a value with an incorrect type leads to a type error, whereas a value that does not match is just not updated using the update *upd* in $pat \Rightarrow upd$.

An interesting question is whether the pattern should be applicable to a value of type $< a : \textbf{bool} >$. Obviously, the value could never match the pattern. Thus, the only question is if it has the correct type. This is not determined by our typing rules. Instead it is necessary to examine the meaning of an update type $< b_1 : \upsilon_1, \cdots, b_n : \upsilon_n >_u$. It describes updates that accept value of variant types that

1. have at least branches b_1,..., b_n with at least the respective types.

   or

2. if they include some branches b_i $(1 \leq i \leq n)$, then the type of the branch must be accepted by updates of type υ_i.

Interestingly, both interpretations are allowed within our typing rules. Finally, it is not difficult to construct a type inference algorithm for the typing rules. Note that an update expression is not well-typed if it has type $\top$.

4 Optimization

The optimization of updates in databases is a complex and intriguing problem. While optimizations on the implementation level, such as caching and concurrency control, have been designed to reduce the expected execution time of udates, algebraic optimizations have not been an issue. However as with query languages, it is possible to identify powerful rewriting rules to improve the cost of executing an update. In this section, we will consider two categories of such rules: The optimization by *rewriting* and *deltafication*.

In order to talk about optimization, we must have a measure of update cost. Let us consider the following: A complex value can be thought of as an edge-labelled tree. Interior nodes represent type constructors – records, sets and variants – while leaf nodes represent base values such as 2 or "Tom". Edges representing attributes in records or choices in variants carry the label of the attribute of choice. The number and labels coming out of a node depends on its type: A variant node, for instance, consists of exactly one edge (subtree) labeled with the choice. In contrast, a set node can have an arbitrary number of unlabelled edges, depending on the number of elements in the set.

Updating a database entails reading and/or updating some of the nodes of the tree representing its complex value. A node can be updated multiple times during one update, since sequential updates on the same value are possible. Since updates often include the evaluation of expressions such as conditions, sub-values (sub-trees) of the databse may have to be retrieved between the updates. As a simplified model, we assume that the cost of an update is determined by the number of updates and the number of evaluations in between. That means in our optimizations we try to reduce the number of overall accesses to the database.

Furthermore, let us assume that each of the constructs in CPL+ is executed in the obvious way. In particular: The construct **if** c **then** upd_1 **else** upd_2 evaluates c and performs either upd_1 or upd_2. The set update $\{su\}$ iterates over the elements of the set and performs the update su on each of them. The complete update e replaces the old tree of the value to be updated by a new tree which is the result of evaluating e.

4.1 Optimizing Update Expression by Rewriting

The semantics of the language CPL+ imply a variety of rewriting rules. Many of them do not reduce the cost of the update directly; it is necessary to apply a series of transformations to restructure and simplify the expression so that a cost reducing transformation can be applied. After presenting the transformation rules, we will illustrate the advantages of the optimizations by a few examples.

Basic Transformations:

$$
\begin{array}{rcl}
(\textbf{if } c \textbf{ then } upd_1 \textbf{ else } upd_2) \; ; \; upd_3 & \Longrightarrow & \textbf{if } c \textbf{ then } (upd_1 \; ; \; upd_3) \textbf{ else } (upd_2 \; ; \; upd_3) \\
upd_1 \; ; \; (\textbf{if } c \textbf{ then } upd_2 \textbf{ else } upd_3) & \Longrightarrow & \textbf{if } c \textbf{ then } (upd_1 \; ; \; upd_2) \textbf{ else } (upd_1 \; ; \; upd_3) \\
upd \; ; \; e & \Longrightarrow & e
\end{array}
$$

The last rule relies on the observation that since there is no pattern between the two updates *upd* and e, the expression e cannot depend on the new value resulting from performing *upd*. Therefore, the first update can be eliminated.

Update Composition: The following rules define transformations for the composition and decomposition of updates along sequences of updates.

$$
\begin{array}{rcl}
(b_1 : upd_1) \; ; \; (b_2 : upd_2) & \Longrightarrow & (b_2 : upd_2) \; ; \; (b_1 : upd_1) \quad (if\, b_1 \neq b_2) \\
(b : upd_1) \; ; \; (b : upd_2) & \Longrightarrow & b : (upd_1 \; ; \; upd_2) \\
(a_1 := upd_1) \; ; \; (a_2 := upd_2) & \Longrightarrow & (a_2 := upd_2) \; ; \; (a_1 := upd_1) \quad (if\, a_1 \neq a_2) \\
(a := upd_1) \; ; \; (a := upd_2) & \Longrightarrow & a := (upd_1 \; ; \; upd_2) \\
\{su_1\} \; ; \; \{su_2\} & \Longrightarrow & \{su_1 \; ; \; su_2\}
\end{array}
$$

The last rule is called *vertical loop fusion*. As an illustration, let us revisit the update example of removing project "Test" from the database:

```
(Projs:={[Name:"Test"] => setdel};
 Persons:={[Info:<Empl:[Projects:{"Test"}]>] => setdel};
 Persons:={Info:= Empl: Projects:={"Test" => setdel}};
 Persons:={[Info:<Manager:[Project:"Test"]>]
           =>  (Info:=Manager: Project:="Void")}
)
```

Performing this update as written means that the set `Persons` is traversed three times. Applying the rule for composition of record updates and set updates, we can transform the update into the following expression:

```
(Projs:={[Name:"Test"] => setdel};
 Persons:={[Info:<Empl:[Projects:{"Test"}]>] => setdel;
           Info:= Empl: Projects:={"Test" => setdel};
           [Info:<Manager:[Project:"Test"]>]
           => (Info:=Manager: Project:="Void")}
)
```

In this optimized query, each person is considered only once.

Filter Promotion: It is often possible to change the order of containing updates in the syntax tree. In particular, we are interested in moving the condition in **if** c **then** upd_1 **else** upd_2 to the outside of the containing update:

$$
\begin{array}{rcl}
a := \textbf{if } c \textbf{ then } upd_1 \textbf{ else } upd_2 & \Longrightarrow & \textbf{if } c \textbf{ then } a := upd_1 \textbf{ else } a := upd_2 \\
b : \textbf{if } c \textbf{ then } upd_1 \textbf{ else } upd_2 & \Longrightarrow & \textbf{if } c \textbf{ then } b : upd_1 \textbf{ else } b : upd_2 \\
\{\textbf{if } c \textbf{ then } upd_1 \textbf{ else } upd_2\} & \Longrightarrow & \textbf{if } c \textbf{ then } \{upd_1\} \textbf{ else } \{upd_2\} \\
\{\textbf{setif } c \textbf{ then } su_1 \textbf{ else } su_2\} & \Longrightarrow & \textbf{if } c \textbf{ then } \{su_1\} \textbf{ else } \{su_2\}
\end{array}
$$

For example, consider an update that removes all employees from project `Proj1` who earn more than \$40000. This entails removing the project `Proj1` from the set of projects of each employee who has a salary bigger than \$40000.

```
@Persons:={\p => Info:= Empl:(\e =>
    Projects:={"Proj1" => (if p.Salary>40000 then setdel)})}
```

Since the condition `p.Salary>40000` does not depend on the project or the employee e, it is possible to delegate the condition to the outside of the tree. This leads to the following update:

```
@Persons:={\p => if p.Salary>40000 then Info:= Empl:(\e =>
   Projects:={"Proj1" => setdel})}
```

As a consequence, only the projects of the persons who have a salary bigger than \$40000 are considered. In the original version, the project set of *each* employee is traversed. Note that for moving **if** c **then** upd_1 **else** upd_2 to the outside of the pattern $\backslash e => upd$, we need to apply a separate rule that requires that the variable e is not used within condition c. As described later, this rule is part of an optimization called *variable delegation.*

Set Updates: There are important simplifications for set updates. For instance, the deletion of an element of a set makes the set updates before or afterwards in the sequence unnecessary. Generally, the following simplification rules can be identified:

$$
\begin{array}{rcl}
su \ ; \ \textbf{setdel} & \Longrightarrow & \textbf{setdel} \\
\textbf{setdel} \ ; \ su & \Longrightarrow & \textbf{setdel} \\
(\textbf{setif } c \textbf{ then } su_1 \textbf{ else } su_2) \ ; \ su_3 & \Longrightarrow & \textbf{setif } c \textbf{ then } (su_1 \ ; \ su_3) \textbf{ else } (su_2 \ ; \ su_3) \\
su_1 \ ; \ (\textbf{setif } c \textbf{ then } su_2 \textbf{ else } su_3) & \Longrightarrow & \textbf{setif } c \textbf{ then } (su_1 \ ; \ su_2) \textbf{ else } (su_1 \ ; \ su_3) \\
\textbf{setins } e_1 \ ; \ \textbf{setins } e_2 & \Longrightarrow & \textbf{setins } e_1 \cup e_2 \\
\{\textbf{setdel}\} \ ; \ \textbf{setins } e & \Longrightarrow & e \\
upd \ ; \ \{\textbf{setdel}\} & \Longrightarrow & \{\textbf{setdel}\}
\end{array}
$$

Pattern Simplification: The previously described optimization rules are only applicable as long as there are no patterns and variable bindings between the expressions. Therefore, a very important part of the optimization process is the analysis and simplification of pattern expressions. Often, patterns can be restructured, relocated or eliminated, as the following rewrite rules show:

$$
\begin{array}{rcl}
pat \Rightarrow ID_{upd} & \Longrightarrow & ID_{upd} \\
\backslash v_1 \Rightarrow (\backslash v_2 \Rightarrow upd) & \Longrightarrow & \backslash v_1 \Rightarrow upd[v_2 \setminus v_1] \\
< b_1 : pat > \Rightarrow b_2 : upd & \Longrightarrow & ID_{upd} \quad (if\, b_1 \neq b_2) \\
< b : pat > \Rightarrow b : upd & \Longrightarrow & b : (pat \Rightarrow upd) \\
< b_1 : pat > \Rightarrow (b_2 : upd_1); upd_2 & \Longrightarrow & < b_1 : pat > \Rightarrow upd_2 \quad (if\, b_1 \neq b_2) \\
a := (\backslash v \Rightarrow upd) & \Longrightarrow & \backslash v' \Rightarrow (\mathbf{let}\ v := \pi_a(v')\ \mathbf{in}\ upd)
\end{array}
$$

It is also possible to decompose record patterns, as in the following example:

$$[a_1 : \backslash v_1, a_2 : \backslash v_2] \Rightarrow upd \quad \Longrightarrow \quad v \Rightarrow (\mathbf{let}\ v_1 := \pi_{a_1}(v)\ \mathbf{in}\ (\mathbf{let}\ v_2 := \pi_{a_2}(v)\ \mathbf{in}\ upd))$$

Most record patterns can be decomposed in this way. However, it is not possible to resolve record patterns that have variant patterns with variable bindings inside its attribute patterns. In this case, it is not possible to generate simple patterns. The decomposition of patterns is based on the following rewriting rules:

$$
\begin{array}{rcl}
pat \Rightarrow upd & \Rightarrow & \backslash nv \Rightarrow \mathcal{P}(pat, upd, nv) \\
 & & \\
\mathcal{P}(\backslash v, upd, pv) & \Rightarrow & \mathbf{let}\ v := pv\ \mathbf{in}\ upd \\
\mathcal{P}(e, upd, pv) & \Rightarrow & \mathbf{if}\ e = pv\ \mathbf{then}\ upd \\
\mathcal{P}([a_1 : pat_1, \cdots, a_n : pat_n], upd, pv) & \Rightarrow & \mathcal{P}(pat_1, \mathcal{P}([a_2 : pat_2, \cdots, a_n : pat_n], upd, pv), \pi_{a_1}(pv)) \\
\mathcal{P}([a : pat], upd, pv) & \Rightarrow & \mathcal{P}(pat, upd, \pi_a(pv)) \\
\mathcal{P}(< b : e >, upd, pv) & \Rightarrow & \mathbf{if}\ (\mathbf{case}\ pv\ \mathbf{of}\ b(v) \Rightarrow (v = e), \cdots \Rightarrow \mathbf{false})\ \mathbf{then}\ upd
\end{array}
$$

The last rule shows that variant patterns are restricted in that they can only contain single constant expressions as their branch pattern. Note this can be generalized so that variant patterns can contain pattern expressions without variable bindings as their branch patterns.

Variable Delegation: It is often possible to delegate variable bindings to lower or higher levels in the syntax tree. This can lead to significant simplifications, since further transformations can be applied. Let us assume that function $f_v()$ denotes the set of all free variables in an expression e, update *upd* or pattern *pat*. The following rules hold:

$$
\begin{array}{lrcl}
v \notin f_v(upd) : & \backslash v \Rightarrow upd & \Leftrightarrow & upd \\
v \notin f_v(upd_2) : & \backslash v \Rightarrow (upd_1\ ;\ upd_2) & \Leftrightarrow & (\backslash v \Rightarrow upd_1)\ ;\ upd_2 \\
v \notin f_v(c) : & \backslash v \Rightarrow (\mathbf{if}\ c\ \mathbf{then}\ upd_1\ \mathbf{else}\ upd_2) & \Leftrightarrow & \mathbf{if}\ c\ \mathbf{then}\ \backslash v \Rightarrow upd_1\ \mathbf{else}\ \backslash v \Rightarrow upd_1 \\
v \notin f_v(c) : & \backslash v \Rightarrow (\mathbf{setif}\ c\ \mathbf{then}\ su_1\ \mathbf{else}\ su_2) & \Leftrightarrow & \mathbf{setif}\ c\ \mathbf{then}\ \backslash v \Rightarrow su_1\ \mathbf{else}\ \backslash v \Rightarrow su_1 \\
v \notin f_v(pat) : & \backslash v \Rightarrow (pat \Rightarrow upd) & \Leftrightarrow & pat \Rightarrow (\backslash v \Rightarrow upd) \\
v \notin f_v(e) : & \backslash v \Rightarrow (\mathbf{let}\ v' := e\ \mathbf{in}\ upd) & \Leftrightarrow & \mathbf{let}\ v' := e\ \mathbf{in}\ \backslash v \Rightarrow upd \\
v \notin f_v(e) : & \backslash v \Rightarrow (\mathbf{setlet}\ v' := e\ \mathbf{in}\ su) & \Leftrightarrow & \mathbf{setlet}\ v' := e\ \mathbf{in}\ \backslash v \Rightarrow su
\end{array}
$$

Based on the previous transformation rules for patterns, it is always possible to eliminate complex patterns. Under the assumption that variant patterns do not contain variable bindings in its branch pattern, it is possible to normalize CPL+ expressions so that they only contain simple variable bindings in front of set updates and variant expressions: $\{\backslash v \Rightarrow upd\}$ and $a := \backslash v \Rightarrow upd$.

Let Elimination: The let expression is syntactic sugar in that the defined variable can be substituted by the respective expression:

$$\begin{array}{rcl} \textbf{let } v := e \textbf{ in } upd & \Leftrightarrow & upd[v \setminus e] \\ \textbf{setlet } v := e \textbf{ in } su & \Leftrightarrow & su[v \setminus e] \end{array}$$

Identity Elimination: Many update expression can be reduced to the identity update if they have the identity update as their component:

$$\begin{array}{rcl@{\qquad}rcl} a := ID_{upd} & \Longrightarrow & ID_{upd} & ID_{upd} \ ; \ upd & \Longrightarrow & upd \\ b : ID_{upd} & \Longrightarrow & ID_{upd} & upd \ ; \ ID_{upd} & \Longrightarrow & upd \\ \textbf{if } c \textbf{ then } ID_{upd} \textbf{ else } ID_{upd} & \Longrightarrow & ID_{upd} & \{ID_{upd}\} & \Longrightarrow & ID_{upd} \\ \textbf{setif } c \textbf{ then } ID_{upd} \textbf{ else } ID_{upd} & \Longrightarrow & ID_{upd} & & & \end{array}$$

Inline Expansion: Consider the update `@Salary:=(s => s*1.2 ; s1 => s1+5000)` which raises the salary of an employee by 20 percent and adds \$5000 to it. This update can obviously be transformed into `@Salary:=(s'=> (s'*1.2)+5000)` reducing the number of updates from two to one. Formally, we interpret the first update as a function and modify the pattern variable of the second update in an appropriate way:

$$(upd_1 \ ; \ \backslash v \Rightarrow upd_2) \ \Longrightarrow \backslash v' \Rightarrow (upd_2[v \mapsto [\![upd]\!](v)])$$

The expression $[\![upd]\!]$ denotes the update function that can be applied to an arbitrary value with the correct type. The difficulty is that it is not obvious when such a replacement improves the cost of the update. Basically, the inline expansion is useful if the entire value is updated. In this case, recomputing the entire value is often cheaper than updating it multiple times.

4.2 Deltafication

In this section we will introduce the notion of "deltafication" as a way of transforming complete updates (expressed in CPL or $\mathcal{NRC}^+$) into CPL+ updates.

Let us consider the employee example again: Increase the salary of "Tom" by \$5000:

```
@Persons:={ \e => if e.Name="Tom" then (Salary:=e.Salary+5000)}
```

Recall that it is also possible to describe the new set of persons as a CPL expression:

```
@Persons:={[Name:n,Age:a,Info:i,Salary: (if (n="Tom") then s+5000 else s)]
   | [Name:\n,Age:\a,Salary:\s,Info:\i] <- Persons};
```

The expression on the right hand side of the record update corresponds to the following $\mathcal{NRC}^+$-expression:

$\bigcup\{\{[\ Name : \pi_{Name}(e), Age : \pi_{Age}(e), Info : \pi_{Info}(e),$
$\quad Salary : \mathbf{if}\ (\pi_{Name}(e) = \text{“}Tom''\text{)}\ \mathbf{then}\ \pi_{Salary}(e) + 5000\ \mathbf{else}\ \pi_{Salary}(e)]\} \mid e \in Persons\}$

For later use we denote this expression as e_{Tom}. Assuming that the number of persons in the set is quite large, the computation of this expression will be expensive, although only one person is actually updated. That means it is desirable to transform the inefficient version involving the computation of the entire set into the update expression that was mentioned first. Note that the rewriting rules for CPL+ given earlier would do nothing to optimize the complete update since it is a CPL expression.

A second, slightly more complex update example is the following:

```
@Persons:={[Info:\i,Name:\n] => Info:=
   case i of <Empl:e> => <Empl:e>,
             <Secr:s> => <Secr: [Manager:
                if n="Sarah" then "Karen" else s.Manager]>,
             <Manager:m> => <Manager: [Project:m.Project,Secretary:
                if n="Karen" then "Sarah" else m.Secretary]>]
}
```

This update expression changes the secretary of manager "Karen" to "Sarah". Furthermore, due to the inverse relationship between manager and secretary the manager of "Sarah" is changed to "Karen". In this example, a complete update is performed on the `Info` attribute of each person. It unnecessarily replaces the complete `Info` attribute value of every person, although only two persons are actually updated. A more efficient (and more concise) version would be the following CPL+ update:

```
@Persons:={\p => Info:=
  (Secr   : if p.Name="Sarah" then Manager:="Karen";
   Manager: if p.Name="Karen" then Secretary:="Sarah"
  )
}
```

In the following, we will present a set of transformation rules that allow the transformation of CPL expressions into more efficient CPL+ update expressions. We will show how to transform the complete updates in both examples into the more efficient delta updates.

Let us consider all CPL-expressions as being converted to $\mathcal{NRC}^+$. The fundamental part of the transformation system is a function $\mathcal{FC}(e_1, e_2)$: **bool** ($\mathcal{FC}$ stands for “FindCondition”) that generates a conditional expression such that if this expression evaluates to **true** under a variable assignment ρ (with the free variables of e_1 and e_2 bound), then the two expressions e_1 and e_2 evaluate to the same value:

$$\forall e_1, e_2, \rho \ : \ \mathcal{N}[\![\mathcal{FC}(e_1, e_2)]\!]\rho \ \Rightarrow \ \mathcal{N}[\![e_1]\!]\rho \equiv \mathcal{N}[\![e_2]\!]\rho$$

Note that the two expression e_1 and e_2 have to have the same type. Since equality of $\mathcal{NRC}^+$ expressions is undecidable, it is not possible to find the *exact* condition under which two $\mathcal{NRC}^+$ expressions denote that same value.

To illustrate the function let us find the condition when the record constructor in the first update example is equal to e:

$\mathcal{FC}(\ [Name : \pi_{Name}(e), Age : \pi_{Age}(e), Info : \pi_{Info}(e), Salary :$
$\quad \mathbf{if}\ (\pi_{Name}(e) = \text{“}Tom''\text{)}\ \mathbf{then}\ \pi_{Salary}(e) + 5000\ \mathbf{else}\ \pi_{Salary}(e)], e)$
$\Longrightarrow \mathbf{not}(\pi_{Name}(e) = \text{“}Tom''\text{)}$

This result is obtained by applying the following sequence of rules:

$$\begin{array}{rcl}
\mathcal{FC}([a_1:e_1,...,a_n:e_n],e) & \equiv & \mathcal{FC}(e_1,\pi_{a_1}(e)) \ \wedge\ \cdots\ \wedge\ \mathcal{FC}(e_n,\pi_{a_n}(e)) \\
\mathcal{FC}(\pi_a(e_1),\pi_a(e_2)) & \equiv & \mathcal{FC}(e_1,e_2) \\
\mathcal{FC}(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, e_4) & \equiv & (e_1\ \wedge\ \mathcal{FC}(e_2,e_4))\ \vee\ (\textbf{not}(e_1)\ \wedge\ \mathcal{FC}(e_2,e_3)) \\
\mathcal{FC}(e,e) & \equiv & \textbf{true} \\
\mathcal{FC}(\cdots,\cdots) & \equiv & \textbf{false}
\end{array}$$

A rule is only applied if the function parameters match the patterns of the rule and the previous rules in the sequence are not applicable. These rules are part of a large rule system that is shown in figure 10. It is important to note that a variety of other rewriting rules are necessary. In our example, for instance, rules for simplification of boolean expressions are needed. Note again that since we are dealing with an undecidable problem we can only approximate the equality condition. That means the two expressions might be equal even though the result of $\mathcal{FC}(e_1,e_2)$ does not indicate that.

We extend the definition of $\mathcal{FC}(e_1,e_2)$ by allowing the use of a special expression $\bigcirc$ at any place within e_2. Two expressions e_1 and e_2 are then considered to be equal if there is a substitute expression e for $\bigcirc$ that makes the two expressions equal:

$$\forall e_1,e_2,\rho\ :\ \mathcal{N}[\![\mathcal{FC}(e_1,e_2)]\!]\rho\ \Rightarrow\ \exists e\ :\ \mathcal{N}[\![e_1]\!]\rho \equiv \mathcal{N}[\![e_2[\bigcirc \setminus e]]\!]\rho$$

Furthermore, a function $\mathcal{FE}(e_1,e_2)$ ($\mathcal{FE}$ stands for "FindExpression") is defined that determines the expression e that matches the symbol $\bigcirc$ in e_2. This expression is only valid if the condition $\mathcal{FC}(e_1,e_2)$ evaluates to **true**.

As shown in figure 10, we extended the set of rules for $\mathcal{FC}(e_1,e_2)$ by the equality $\mathcal{FC}(\cdots,\bigcirc) \equiv \textbf{true}$.

This rule system consists of the most important rewriting rules. However, it is possible to identify more, probably more complex rules for evaluating $\mathcal{FC}(\cdot,\cdot)$. Using more complex structures, it is possible to obtain rules for $\bigcup\{e_1 \mid v \in e_2\}$ and other constructs. They are, however, beyond the scope of this paper. The rules for records and variants in figure 10 are an outcome of the following equivalences:

$$\boxed{\begin{array}{rcl}
\pi_{a_i}([a_1:e_1,\cdots,a_n:e_n]) & \equiv & e_i \\
[a_1:\pi_{a_1}(e),\cdots,a_n:\pi_{a_n}(e)] & \equiv & e \\
\textbf{case } e \textbf{ of } b_1(v_1) \Rightarrow e,\cdots,b_n(v_n) \Rightarrow e & \equiv & e \\
\textbf{case } e \textbf{ of } b_1(v_1) \Rightarrow < b_1 : v_1 >,\cdots,b_n(v_n) \Rightarrow < b_n : v_n > & \equiv & e
\end{array}}$$

In order to evaluate project expressions correctly, we use special construct for partial records: $[a_1:e_1,...,a_n:e_n,\Box]$. Let us consider the evaluation of

$$\mathcal{FC}(\pi_{a_1}(\textbf{if } c \textbf{ then } [a_1:1,a_2:2] \textbf{ else } [a_1:1,a_2:3]),1)$$

Clearly, this should yield **true**. Partial records allow us to resolve this kind of expressions, as one can easily verify by applying the rules to the example:

$$\begin{array}{l}
\mathcal{FC}(\pi_{a_1}(\textbf{if } c \textbf{ then } [a_1:1,a_2:2] \textbf{ else } [a_1:1,a_2:3]),1) \\
\equiv \mathcal{FC}(\textbf{if } c \textbf{ then } [a_1:1,a_2:2] \textbf{ else } [a_1:1,a_2:3],[a_1:1,\Box]) \\
\equiv (c \wedge \mathcal{FC}([a_1:1,a_2:2],[a_1:1,\Box])) \vee (\textbf{not}(c) \wedge \mathcal{FC}([a_1:1,a_2:3],[a_1:1,\Box]))
\end{array}$$

$$
\begin{array}{rcl}
\mathcal{FC}(\underline{c}_1, \underline{c}_2) & \equiv & (\underline{c}_1 = \underline{c}_2) \\
\mathcal{FC}(v, \underline{c}) & \equiv & (v = \underline{c}) \\
\mathcal{FC}(v_1, v_2) & \equiv & (v_1 = v_2) \\
\mathcal{FC}(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, e_4) & \equiv & (e_1 \ \wedge\ \mathcal{FC}(e_2, e_4)) \ \vee\ (\textbf{not}(e_1) \ \wedge\ \mathcal{FC}(e_2, e_4)) \\
\mathcal{FC}(\pi_a(e), e') & \equiv & \mathcal{FC}(e, [a : e', \Box]) \\
\mathcal{FC}([a_1 : e_1, ..., a_n : e_n], e) & \equiv & \mathcal{FC}(e_1, \pi_{a_1}(e)) \ \wedge\ \cdots\ \wedge\ \mathcal{FC}(e_n, \pi_{a_n}(e)) \\
\mathcal{FC}(e, \pi_{a_i}([a_1 : e_1, ..., a_n : e_n])) & \equiv & \mathcal{FC}(e, e_i) \\
\mathcal{FC}(e, \pi_{a_i}([a_1 : e_1, ..., a_n : e_n, \Box])) & \equiv & \mathcal{FC}(e, e_i) \\
\mathcal{FC}(e, \pi_b([a_1 : e_1, ..., a_n : e_n, \Box])) & \equiv & \textbf{true} \\
\mathcal{FC}(< b : e >, < b : e' >) & \equiv & \mathcal{FC}(e, e') \\
\mathcal{FC}(\textbf{case } e \textbf{ of } b_1(v_1) \Rightarrow e_1 \ , ..., b_n(v_n) \Rightarrow e_n, e') & \equiv & \textbf{case } e \textbf{ of} \\
 & & b_1(v_1) \Rightarrow (\mathcal{FC}(e_1, e') \ \vee \\
 & & \qquad (\mathcal{FC}(e_1, < b_1 : v_1 >) \ \wedge\ \mathcal{FC}(e, e')) \\
 & & , ..., \\
 & & b_n(v_n) \Rightarrow (\mathcal{FC}(e_n, e') \ \vee \\
 & & \qquad (\mathcal{FC}(e_n, < b_n : v_n >) \ \wedge\ \mathcal{FC}(e, e')) \\
\mathcal{FC}(\{\}, \{\}) & \equiv & \textbf{true} \\
\mathcal{FC}(\{e\}, \{e'\}) & \equiv & \mathcal{FC}(e, e') \\
\mathcal{FC}(e_1 \cup e_2, e_1' \cup e_2') & \equiv & (\mathcal{FC}(e_1, e_1') \ \wedge\ \mathcal{FC}(e_2, e_2')) \ \vee \\
 & & (\mathcal{FC}(e_1, e_2') \ \wedge\ \mathcal{FC}(e_2, e_1')) \\
\mathcal{FC}(\cdots, \bigcirc) & \equiv & \textbf{true} \\
\mathcal{FC}(\cdots, \cdots) & \equiv & \textbf{false}
\end{array}
$$

Figure 10: Rules for $\mathcal{FC}(\cdot, \cdot)$

$$
\begin{array}{l}
\equiv (c \wedge (\mathcal{FC}(1, \pi_{a_1}([a_1 : 1, \Box])) \vee \mathcal{FC}(2, \pi_{a_2}([a_1 : 1, \Box])))) \vee \\
\quad (\textbf{not}(c) \wedge (\mathcal{FC}(1, \pi_{a_1}([a_1 : 1, \Box])) \vee \mathcal{FC}(3, \pi_{a_2}([a_1 : 1, \Box])))) \\
\equiv (c \wedge (\mathcal{FC}(1, 1) \vee \textbf{true})) \vee (\textbf{not}(c) \wedge (\mathcal{FC}(1, 1) \vee \textbf{true})) \\
\equiv c \vee \textbf{not}(c) \\
\equiv \textbf{true}
\end{array}
$$

Based on these functions, an algorithm for deltafication has been developed. Let us illustrate the main idea of deltafication using the expression $\bigcup\{e_1 \mid v \in e_2\}$. Deltafication means basically to find out how the expression e_2 is changed through the expression. We can make the following observation: An element of e_2 gets deleted if $\mathcal{FC}(e_1, \{\})$ holds. It gets modified if e_1 yields a singleton set, i.e. $\mathcal{FC}(e_1, \{\bigcirc\})$ is true. The modified expression is obtained by $\mathcal{FE}(e_1, \{\bigcirc\})$. If none of the two conditions is true, then the element has to be replaced by expression e_1.

Let us now consider the deltafication function $\Delta(e_1, e_2)$. It identifies the changes between e_1 and e_2 and generates an appropriate CPL+ update expression that, if applied to e_1, yields e_2:

$$[\![\Delta(e_1, e_2)]\!]_\tau \rho(e_1) \equiv \mathcal{N}[\![e_2]\!]\rho$$

For our update example, we can rewrite the update in the following way:

$$@Persons := e_{Tom} \quad \Longrightarrow \quad @Persons := \Delta(Persons, e_{Tom})$$

The function $\Delta(e_1, e_2)$ is defined recursively on the structure of e_2. The previous observation about the deltafication of $\bigcup\{e_1 \mid v \in e_2\}$ is reflected in the following rule:

$$\begin{aligned}\Delta(e_2, \bigcup\{e_1 \mid v \in e_2\}) \quad \equiv \quad & \{v \Rightarrow \textbf{setif } \mathcal{FC}(e_1, \{\}) \textbf{ then setdel else} \\ & \quad \textbf{setif } \mathcal{FC}(e_1, \{\bigcirc\}) \textbf{ then } \Delta(v, \mathcal{FE}(e_1, \{\bigcirc\})) \textbf{ else setrepl } e_1)\}\end{aligned}$$

This rule is a special case of a larger system of rules that is shown in figure 11. These rules allow the deltafication of a wide range of $\mathcal{NRC}^+$ expressions.

$$\begin{array}{rcl}
\Delta(v, v) & \Longrightarrow & ID_{upd} \\
\Delta(e_1, \textbf{if } c \textbf{ then } e_2 \textbf{ else } e_3) & \Longrightarrow & \textbf{if } c \textbf{ then } \Delta(e_1, e_2) \textbf{ else } \Delta(e_1, e_3) \\
\\
\Delta(e_1, \bigcup\{e_2 \mid v \in e_3\}) & \Longrightarrow & \Delta(e_1, e_3) \ ; \\
& & \{\backslash v \Rightarrow \textbf{setif } \mathcal{FC}(e_2, \{\}) \textbf{ then setdel else} \\
& & \quad \textbf{setif } \mathcal{FC}(e_2, \{\bigcirc\}) \textbf{ then } \Delta(v, \mathcal{FE}(e_2, \{\bigcirc\})) \\
& & \qquad \textbf{else setrepl } e_2)\} \\
\Delta(\cdots, \{\}) & \Longrightarrow & \{\textbf{setdel}\} \\
\Delta(e_1, e_2 \cup e_3) & \Longrightarrow & \Delta(e_1, e_2); \textbf{setins } e_3 \\
\\
\Delta(e, [a_1 : e_1, ..., a_n : e_n]) & \Longrightarrow & (a_1 := \Delta(\pi_{a_1}(e), e_1); \cdots ; a_n := \Delta(\pi_{a_n}(e), e_n)) \\
\Delta(\pi_a(e_1), \pi_a(e_2)) & \Longrightarrow & \Delta(e_1, e_2) \\
\Delta(e_1, \pi_a(e_2)) & \Longrightarrow & \textbf{if } \mathcal{FC}(e_2, [a : \bigcirc, \square]) \\
& & \textbf{then } \Delta(\mathcal{FE}(e_2, [a : \bigcirc, \square]), e_1) \textbf{ else } \pi_a(e_2) \\
\\
\Delta(e, \textbf{case } e \textbf{ of } b_1(v_1) \Rightarrow e_1 \ , ..., b_n(v_n) \Rightarrow e_n) & \Longrightarrow & b_1 : \backslash v_1 \Rightarrow (\textbf{if } \mathcal{FC}(e_1, < b_1 : \bigcirc >) \\
& & \quad \textbf{then } \Delta(v_1, \mathcal{FE}(e_1, < b_1 : \bigcirc >)) \textbf{ else } e_1); \\
& & \vdots \\
& & b_n : \backslash v_n \Rightarrow (\textbf{if } \mathcal{FC}(e_n, < b_n : \bigcirc >) \\
& & \quad \textbf{then } \Delta(v_n, \mathcal{FE}(e_n, < b_n : \bigcirc >)) \textbf{ else } e_n) \\
\\
\Delta(\cdots, e) & \Longrightarrow & e
\end{array}$$

Figure 11: Rules for $\Delta(\cdot, \cdot)$

Let us now consider the expression $\Delta(Persons, e_{Tom})$. e_{Tom} is of the form $\bigcup\{e_1 \mid v \in e_2\}$ with $e_2 = Persons$ so that the previous rule can be applied. Obviously, the following equalities hold:

$$\begin{aligned}&\mathcal{FC}(\{[\ Name : \pi_{Name}(e), Age : \pi_{Age}(e), Info : \pi_{Info}(e), \\ &\quad Salary : \textbf{if } (\pi_{Name}(e) = \text{“}Tom'') \textbf{ then } \pi_{Salary}(e) + 5000 \textbf{ else } \pi_{Salary}(e)]\}, \{\}) = \textbf{false}\end{aligned}$$

and

$$\mathcal{FC}(\{[\ Name : \pi_{Name}(e), Age : \pi_{Age}(e), Info : \pi_{Info}(e),$$
$$Salary : \textbf{if } (\pi_{Name}(e) = \text{“}Tom\text{''}) \textbf{ then } \pi_{Salary}(e) + 5000 \textbf{ else } \pi_{Salary}(e)]\}, \{\bigcirc\}) = \textbf{true}$$

with

$$\mathcal{FE}(\{[\ Name : \pi_{Name}(e), Age : \pi_{Age}(e), Info : \pi_{Info}(e),$$
$$Salary : \textbf{if } (\pi_{Name}(e) = \text{“}Tom\text{''}) \textbf{ then } \pi_{Salary}(e) + 5000 \textbf{ else } \pi_{Salary}(e)]\}, \{\bigcirc\})$$
$$= [\ Name : \pi_{Name}(e), Age : \pi_{Age}(e), Info : \pi_{Info}(e),$$
$$Salary : \textbf{if } (\pi_{Name}(e) = \text{“}Tom\text{''}) \textbf{ then } \pi_{Salary}(e) + 5000 \textbf{ else } \pi_{Salary}(e)]$$

Therefore, we can rewrite:

$$\Delta(Persons, e_{Tom}) = \{e \Rightarrow \textbf{setif false then setdel else (setif true then}$$
$$\Delta(e, [\ Name : \pi_{Name}(e), Age : \pi_{Age}(e), Info : \pi_{Info}(e),$$
$$Salary : \textbf{if } (\pi_{Name}(e) = \text{“}Tom\text{''}) \textbf{ then } \pi_{Salary}(e) + 5000 \textbf{ else } \pi_{Salary}(e)])$$
$$\textbf{else } \cdots\}$$

Using the rule in figure 11 for resolving $\Delta(e, [a_1 : e_1, ..., a_n : e_n])$ and the rules for simplifying $CPL+$ expressions, the following result is finally obtained:

$$\Delta(Persons, e_{Tom})$$
$$\equiv$$
$$\{e \Rightarrow (\ Name := \Delta(\pi_{Name}(e), \pi_{Name}(e));$$
$$Info := \Delta(\pi_{Info}(e), \pi_{Info}(e));$$
$$Salary := \Delta(\pi_{Salary}(e),$$
$$\textbf{if } (\pi_{Name}(e) = \text{“}Tom\text{''}) \textbf{ then } \pi_{Salary}(e) + 5000 \textbf{ else } \pi_{Salary}(e)))\}$$
$$\equiv$$
$$\{e \Rightarrow (Salary := \textbf{if } (\pi_{Name}(e) = \text{“}Tom\text{''}) \textbf{ then } \Delta(\pi_{Salary}(e), \pi_{Salary}(e) + 5000) \textbf{ else } \pi_{Salary}(e))\}$$
$$\equiv$$
$$\{e \Rightarrow \textbf{setif } (\pi_{Name}(e) = \text{“}Tom\text{''}) \textbf{ then } Salary := \pi_{Salary}(e) + 5000\}$$

Let us recall the second update example, which changes the secretary of manager "Karen" to "Sarah". In this example, a complete update is performed on the `Info` attribute of each person. A very important requirement for the application of deltafication rules is the simplification of patterns. Using the rewriting rules of CPL+ the pattern `[Info:\i,Name:\n] => Info:=...` can be replaced by `\p => Info :=...` where `i` and `n` are replaced by `p.Info` and `p.Name`, respectively:

```
@Persons:={\p => Info:=
   case p.Info of <Empl:e> => <Empl:e>,
            <Secr:s> => <Secr: [Manager:
               if p.Name="Sarah" then "Karen" else s.Manager]>,
            <Manager:m> => <Manager: [Project:m.Project,Secretary:
               if p.Name="Karen" then "Sarah" else m.Secretary]>]
}
```

The CPL expression `case p.Info...`, which updates the `Info` attribute, is transformed into the following $\mathcal{NRC}^+$ expression:

$$\textbf{case } \pi_{Info}(p) \textbf{ of } Empl(e) \Rightarrow < Empl : e >,$$

$$
\begin{array}{l}
Secr(s) \Rightarrow < Secr : [Manager : \textbf{if } \pi_{Name}(p) = "Sarah" \textbf{ then } "Karen" \textbf{ else } \pi_{Manager}(s)] >, \\
Manager(m) \Rightarrow < Manager : [Project : \pi_{Project}(m), Secretary : \\
\qquad\qquad\qquad \textbf{if } \pi_{Name}(p) = "Karen" \textbf{ then } "Sarah" \textbf{ else } \pi_{Secretary}(m)] >
\end{array}
$$

Let us denote this $\mathcal{NRC}^+$ expression as e_{Karen}. Note that this expression has only one free variable, p, which is bound within the containing CPL+ pattern expressions. It is easily observable that the value updated by the $\mathcal{NRC}^+$ expression is $\pi_{Info}(p)$. Therefore, we can rewrite the CPL+ expression in the following way:

$$
@Persons := \{\backslash p \Rightarrow Info := e_{Karen}\} \quad \Longrightarrow \quad @Persons := \{\backslash p => Info := \Delta(\pi_{Info}(p), e_{Karen})\}
$$

It is now possible to apply the deltafication rules listed in figure 11:

$$
\begin{array}{l}
\Delta(\pi_{Info}(p), e_{Karen}) \\
\Longrightarrow \\
\quad Empl : \backslash e \Rightarrow \textbf{if true then } \Delta(e, e) \textbf{ else } < Empl : e >; \\
\quad Secr : \backslash s \Rightarrow \textbf{if true then } \Delta(s, [Manager : \\
\qquad\qquad\qquad \textbf{if } \pi_{Name}(p) = "Sarah" \textbf{ then } "Karen" \textbf{ else } \pi_{Manager}(s)]) \textbf{ else } \cdots; \\
\quad Manager : \backslash m \Rightarrow \textbf{if true then } \Delta(m, [Project : \pi_{Project}(m), \\
\qquad\qquad\qquad\quad Secretary : \textbf{if } \pi_{Name}(p) = "Karen" \textbf{ then } "Sarah" \textbf{ else } \pi_{Secretary}(m)]) \textbf{ else } \cdots; \\
\Longrightarrow \\
\quad Secr : \backslash s \Rightarrow (Manager := \Delta(\pi_{Manager}(s), \textbf{if } \pi_{Name}(p) = "Sarah" \textbf{ then } "Karen" \textbf{ else } \pi_{Manager}(s))); \\
\quad Manager : \backslash m \Rightarrow (\ Project := \Delta(\pi_{Project}(m), \pi_{Project}(m)); \\
\qquad\qquad\qquad\quad Secretary := \Delta(\ \pi_{Secretary}(m), \\
\qquad\qquad\qquad\qquad\qquad \textbf{if } \pi_{Name}(p) = "Karen" \textbf{ then } "Sarah" \textbf{ else } \pi_{Secretary}(m) \\
\Longrightarrow \\
\quad Secr : \backslash s \Rightarrow (Manager := \\
\qquad \textbf{if } \pi_{Name}(p) = "Sarah" \\
\qquad \textbf{then } \Delta(\pi_{Manager}(s), "Karen") \textbf{ else } \Delta(\pi_{Manager}(s), \pi_{Manager}(s)) \\
\quad Manager : \backslash m \Rightarrow (Secretary := \\
\qquad \textbf{if } \pi_{Name}(p) = "Karen" \\
\qquad \textbf{then } \Delta(\pi_{Secretary}(m), "Sarah") \textbf{ else } \Delta(\pi_{Secretary}(m), \pi_{Secretary}(m)) \\
\Longrightarrow \\
\quad Secr : \textbf{if } \pi_{Name}(p) = "Sarah" \textbf{ then } Manager := "Karen"; \\
\quad Manager : \textbf{if } \pi_{Name}(p) = "Karen" \textbf{ then } Secretary := "Sarah"
\end{array}
$$

This leads to the already mentioned, more concise and efficient update expression.

5 Conclusions

In this paper we presented an update language for complex value databases based on the functional query language CPL. The update language, CPL+, allows the intuitive and concise specification of updates and admits a wide range of optimizations.

Two forms of optimizations were presented: Rewriting update expressions and deltafication. Deltafication is special type of rewriting rule that relies on the analysis of query expressions. Most often, only a small part of the database is actually changed during an update. Therefore, converting complete updates (query expressions) into CPL+ expressions that only update the changing parts of a value is a powerful optimization tool. A representative set of rewrite rules were given, and their effectiveness illustrated using some typical update examples.

A fundamental issue in designing CPL+ was the contradiction between evaluation in functional programming languages and the imperative character of stateful functions like assignment. We solved this problem by introducing a separate top-level language construct for update. The extension was based on an imperative paradigm with notions of sequential execution, context, and side effects.

The problem of incorporating stateful functions in functional languages has been well studied. I/O systems and arrays are two examples where side effects are an essential cornerstone of the system. The concept of monads as a method for implementing side effects in functional programming languages was discussed by Wadler [1], Moggi [23] and more recently in [7]. A wide range of other theoretical concepts have also been developed (*Linear Logic* [5], *Persistant Functional Language* [29], *Variable Type Logic of Effects* [22], etc.) Some of the proposed frameworks have been implemented in languages such as Haskell [17]. Based on the work of Reynolds on idealized Algol [26], different type systems and derivation of the lambda calculus that include imperative statements have also been developed [30, 31, 21]. Similar to the work presented in this paper, they separate stateful and stateless functions, introducing multiple layers of types.

While the logical specification of updates in relational and deductive databases has been studied (e.g. [12, 13, 25, 2, 4, 24]), the design, implementation and optimization of update languages in object-oriented and complex value databases has received little attention. The interaction between updates and complex type systems on the language level and the potential for optimizations was recently studied by Hull [20], where a notion of "hypothetical updates" on complex values as a database programming language concept was given. As with CPL+, delta update primitives for specific complex types, such as records and sets, were provided along with optimization rules. Hypothetical updates, however, do only produce virtual states that can be used by subsequent updates. That means the semantics of updates is different.

The language CPL+ and its optimization rules (including deltafication) have been partially implemented, although no prototype is available yet. Various interesting practical and theoretical aspects remain to be investigated. The specification of an execution model with a more detailed cost analysis to allow the dynamic optimization of updates should be studied. The issue of transaction primitives with a reconsideration of optimizations should be studied. Since CPL is currently being used for querying multiple, heterogeneous database systems [15], the issue of what updates across databases mean and how to optimize them should also be addressed. This raises the interesting questions about the design of update interfaces between the databases and the CPL+ update processor. Lastly, embedding the core language in application languages and visualizations tools is, as always, useful for providing a user-friendly environment.

Acknowledgments: We would like to thank Peter Buneman for his original ideas about the update language and Val Tannen for many helpful discussions.

References

[1] *Imperative Functional Programming*, 1993.

[2] S. Abiteboul. Updates, a new frontier. In *Second Conference on Database Theory*, pages 1–18. Springer, 1988.

[3] S. Abiteboul and P. Kanellakis. Query languages for complex object databases. *SIGACT News*, 21(3):9–18, 1990.

[4] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, August 1991.

[5] Samon Abramsky. Computational interpretation of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[6] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O_2 object-oriented database system. In *Proceedings of 2nd International Workshop on Database Programming Languages*, pages 122–138. Morgan Kaufmann, 1989.

[7] N. Beton and P. Wadler. Linear logic, monads and the lambda calculus. In *Proceedings of 11th IEEE Symposium on Logic in Computer Science*, New Brunswick, New Jersey, July 1996.

[8] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 9–19. Morgan Kaufmann, August 1991. Also available as UPenn Technical Report MS-CIS-92-17.

[9] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In J. Biskup and R. Hull, editors, *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 1992. Available as UPenn Technical Report MS-CIS-92-47.

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

[11] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Mateo, California, 1996.

[12] W. Chen. Declarative updates of relational databases. *ACM Transactions on Database Systems*, 20(1):42–70, 1995.

[13] E. Bertino D. Montesi and M. Martelli. Transactions and updates in deductive databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(5):784–797, 1997.

[14] S.B. Davidson, C. Hara, and L Popa. Querying an object-oriented database using CPL. In *Proceedings of the Brazilian Symposium on Databases*, October 1997.

[15] Susan Davidson, Christian Overton, Val Tannen, and Limsoon Wong. Biokleisli: A digital library for biomedical researchers. *Journal of Digital Libraries*, 1(1), November 1996.

[16] R. Zicari F. Ferrandina, T. Meyer, G. Ferran, and J. Madec. Schema and database evolution in the o_2 object database system. In *Proceedings of the 21th International Conference on VLDB*, pages 170–181, Zrich, Switzerland, September 1995.

[17] Joseph H. Fasel, Paul Hudak, Simon Peyton-Jones, and Philip Wadler. The functional programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[18] Stephane Grumbach and Victor Vianu. Tractable query languages for complex object databases. Technical Report 1573, INRIA, Rocquencourt BP 105, 78153 Le Chesnay, France, December 1991. Extended abstract appeared in PODS 91.

[19] H. Liefke and S.B. Davidson. Updating complex value databases. Technical Report MS-CIS-98-06, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pa 19104, 1998.

[20] R. Hull M. Doherty and M. Rupawalla. Structures for manipulating proposed updates in object-oriented databases. In *SIGMOD Conf*, pages 306–317, 1996.

[21] D. Rabin M. Ordersky and P. Hudak. Call by name, assignment, and the lambda calculus. In *Conference Records of the 20th ACM Symposium on Principles of Programming Languages*, pages 43–56, Charleston, South Carlina, January 1993.

[22] I. Mason and C. Talcott. Reasoning about object systems in vtloe. *Journal of Foundations of Computer Science*, 6(3):265–298, 1995.

[23] E. Moggi. Computational lambda calculus and monads. In *Proceedings of 4th IEEE Symposium on Logic in Computer Science*, California, June 1989.

[24] J.D. Ullman R. Fagin and M.Y. Yardi. Updating logical databases. In *Proceedings of the ACM Symposium on Principles of Database Systems*. Springer, 1988.

[25] R. Reiter. On specifying database updates. *Journal of Logic Programming*, 25(1):53–91, 1995.

[26] J.C. Reynolds. The essence of algol. In *Proceedings of ACM Symposium on Algorithmic Languages*, pages 345–372, North Holland, 1981.

[27] John F. Roddick. Schema evolution in database systems — An annotated bibliography. *SIGMOD Record*, 21(4):35–40, December 1992.

[28] Andrea H. Skarra and Stanley B. Zdonik. Type evolution in an object oriented database. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object Oriented Programming*, pages 392–415. MIT Press, Cambridge, Massachusetts, 1987.

[29] Carol Small and Alexandra Poulovassilis. An overview of PFL. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 96–110. Morgan Kaufmann, August 1991.

[30] J.C. Springer. *Implementation of Functional Languages with State*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[31] D. Sutton and C. Small. Extending functional database languages to update completeness. July 1995.

[32] P. W. Trinder. Comprehensions, a query notation for DBPLs. In *Proceedings of 3rd International Workshop on Database Programming Languages, Nahplion, Greece*, pages 49–62. Morgan Kaufmann, August 1991.

[33] P. W. Trinder and P. L. Wadler. Improving list comprehension database queries. In *Proceedings of TENCON'89, Bombay, India*, pages 186–192, November 1989.

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

[35] Limsoon Wong. *Querying Nested Collections*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. Available as University of Pennsylvania IRCS Report 94-09.