Specifying Updates in Biomedical Databases *

Hartmut Liefke and Susan B. Davidson
Department of Computer and Information Science
University of Pennsylvania
200 South 33rd Street, Philadelphia, PA 19104-6389
liefke@seas.upenn.edu and susan@cis.upenn.edu

Abstract

Many of the publicly available biomedical data sources – such as Genbank and SwissProt – are not stored in traditional databases but in a variety of file formats (e.g. ASN.1 and EMBL). The data is complex, involving deeply nested structures. While query languages for such data have been well-studied [1, 16, 2, 10], the issue of updating such databases has not. The need for a concise update language is critical since the changes to the data are typically very small when compared to the entire value.

Starting with a query language called Collection Programming Language (CPL), we describe an extension called CPL+ which provides an intuitive framework for updates on complex values. We illustrate the language using examples and present various optimization that can substantially improve the performance of complex updates.

1 Introduction

Many of the publicly available biomedical data sources – such as Genbank and SwissProt – are not stored in traditional databases but in a variety of file formats (e.g. ASN.1 and EMBL). These formats have been adopted for several reasons. First, the data is complex and not easy to represent in a relational DBMS. Typical structures include sequential data (lists), deeply nested record structures, and variant types. This complexity would argue for the use of object-oriented database systems, but these have not been successful within the community because of the constant need for database restructuring [9]. Second, formatted files are easily accessed from languages such as Fortran and C, and a number of useful software programs exist that work with these files. Third, the files and associated retrieval packages are available for a variety of platforms.

*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.

An example of a Genbank entry is shown in Figure 1. It is a record type, whose fields are complex. Keywords, for example, can be thought of as a set of strings. Reference can be thought of as a list of records (indicated by the indentation). Furthermore, a list of references (with information about authors, title, journal title, etc.) is provided. (For simplicity, the entry in Fig. 1 has only one reference.) Features can be thought of as a set of records with key, location, qualifiers and description fields. The key field can take on one of several different values (e.g. source, gene, allele, CDS, etc). The location field can be an interval (as shown in this example), an integer, a reference, or other types.

Although the formats themselves have a number of advantages, the information retrieval systems associated with them are quite limited. Typically, only complete entries (such as the one shown in Figure 1) can be retrieved based on path-based indexed searches. However, pruning of the entry tree (the analog to relational projection) or restructuring of entries which involve nesting,unnesting or joins cannot be specified. For this purpose, a general purpose query language for complex values must be used (see [1, 16, 2, 10] for a number of such proposals). Research at Penn has produced a powerful language for querying and transforming complex structured data called the Collection Programming Language (CPL) [16, 3, 4] which has been implemented in a system called Kleisli [8]. CPL is being used in various projects involving querying and integrating biomedical data sources at the Penn Center for Bioinformatics as well as at SmithKline Beecham Pharmaceuticals. Chief among the contributions of the Kleisli system are drivers that connect multiple, heterogeneous biomedical data sources and optimization techniques that extend many known techniques and derive new techniques for these more complex data types.

However, the issue of updating complex value databases has not been studied. For example, a standard for object-oriented query languages is given in [6] (OQL), but there is no mention of updates apart from the notion of a transaction and methods. Updates within biomedical databases are cur-

```
LOCUS       HUMNF1AB     8959 bp    mRNA            PRI       07-JAN-1995
DEFINITION  Human type 1 neurofibromatosis protein mRNA, complete cds.
ACCESSION   M82814
NID         g189164
KEYWORDS    NF1 GAP-related protein; NF1 gene product; type 1 neurofibromatosis
            protein.
SOURCE      Homo sapiens cDNA to mRNA.
  ORGANISM  Homo sapiens
            Eukaryotae; mitochondrial eukaryotes; Metazoa; Chordata;
            Vertebrata; Eutheria; Primates; Catarrhini; Hominidae; Homo.
REFERENCE   1  (bases 1 to 8959)
  AUTHORS   Collins,F.S., Andersen,L.B., Marchuk,D.A. and Wallace,M.R.
  TITLE     Type 1 neurofibromatosis gene: correction [letter]
  JOURNAL   Science 250 (4988), 1749 (1990)
  MEDLINE   91102559
FEATURES             Location/Qualifiers
     source          1..8959
                     /organism="Homo sapiens"
                     /db_xref="taxon:9606"
     gene            212..8668
                     /gene="NF1"
BASE COUNT     2598 a   2024 c   1888 g   2449 t
ORIGIN
        1 ccccagcctc cttgccaacg ccccctttcc ctctcccct cccgctcggc gctgaccccc
       61 catccccacc cccgtgggaa cactgggagc ctgcactcca cagaccctct ccttgcctct
...
```

Figure 1. A Genbank Entry

rently performed manually; trusted maintainers pull up an entry on a screen and make changes. There is no notion of a set-oriented update for complex value data as, for example, is present for the relational model in languages such as SQL-92.

Consider the following update example: If a new prefix of sequence information is found for an already existing Genbank entry, then the starting position of existing features must be shifted up by the length of the sequence prefix. Given a flat relational schema for Genbank entries containing tables for entries, features, references, we could write the following update:

```
UPDATE Feature F
SET    F.Location= F.Location + prefix_size
WHERE  F.EntryId= 'M82814'
```

The syntax is terse and set-oriented, specifying only the pieces of the database that are to be altered. In the complex valued representation, the update cannot be specified in a similar fashion and it must be performed manually. This is tedious to do when there are large numbers of features for the updated entry. Furthermore, due to the deeply nested structures in complex value databases, the need for a terse specification of updates is even greater than in relational databases.

In this paper, we address the problem of providing a concise and optimizable update language for complex valued data by building on CPL. In Section 2, we formalize the complex data model and illustrate CPL through a series of examples. We extend CPL at the top level with constructs for updates. The update language, CPL+, is presented in Section 3. Reminiscent of OQL, it provides a concise yet intuitive syntax for complex value updates. Furthermore, an internal, more manipulable calculus called Complex Update CAlculus ($\mathcal{CUCA}$) is presented as a basis for optimizations.

In Section 4, we describe two categories of optimizations for CPL+: Rewrite rules and deltafication. Deltafication generates more efficient CPL+ updates from complete updates. As a result of these transformations, many complex updates can be simplified and performed more efficiently on the database.

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 collection types – sets, bags, and lists – together with record and variant types [5, 16]. For the purposes of this paper we will restrict our attention to bags (i.e. multi-sets) as the only collection type – thus, we leave the problem of duplicate removal during update processing for future work. The set of CPL types is given by the syntax:

$$\tau ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{real} \mid \mathbf{string} \mid \mathsf{bag}(\tau) \mid ... \mid \mathsf{struct}(a_1 : \tau_1, ..., a_n : \tau_n) \mid \mathsf{var}(b_1 : \tau_1, ..., b_n : \tau_n)$$

The types **bool**, **int**, **real**, **string**, etc. are built-in base types. $\mathsf{struct}(a_1 : \tau_1, \ldots, a_n : \tau_n)$ constructs record types from the types $\tau_1, \ldots, \tau_n$ with attributes $a_1, \ldots, a_n$. $\mathsf{var}(b_1 : \tau_1, \ldots, b_n : \tau_n)$ constructs variant types from the types $\tau_1, \ldots, \tau_n$ with branch labels $b_1, \ldots, b_n$; $\mathsf{bag}(\tau)$ constructs a bag of element type τ.

In a genetic database, we could use the following (simplified) type Entry to represent sequence data:

```
struct( ID: int, Org: string, Seq: string,
        Mod: struct(Year: int, Month: int, Day: int),
        Feats: bag(struct(Type: string, FOrg: string,
             Loc: var( Reg: struct(From: int, To: int),
                  Pos: int))))
```

In this type, each entry has an identifier (e.g., an accession number[1]), an organism name, and a sequence string. Furthermore, the date of the last change and the set of features are part of the entry. For simplicity, we omit many other important attributes, such as the set of references, the set of keywords, etc.

Features store information about specific regions of the DNA sequence. In Genbank, several feature types are used, e.g. *intron*, *exon*, *allele*, *CDS* (Coding Sequence), or *source*, each of them with different characteristics. For simplicity, the feature type is represented in string attribute Type. Features can have references to other features, entries, or organisms. Depending on the type of feature, this could be the location of the feature's origin, a reference to a similar feature, etc. For our purposes, we only store the organism in attribute Organism.

The location information for GenBank features can be quite complex. For example, features can constitute multiple intervals on a gene sequence. For simplicity, we only distinguish between two types of locations: regions (represented by a pair of integers) and positions (represented by a single integer).

Values in CPL are explicitly constructed as follows: $\mathsf{struct}(a_1 : e_1, \ldots, a_n : e_n)$ for records, giving values of the appropriate type for each of the attributes; $\mathsf{var}(a : e)$ for variants, giving a value of the appropriate type for one of the labels; and $\mathsf{bag}(e_1, \ldots, e_n)$ for bags. For example, a value representing parts of the entry in Figure 1 could have the following form:

```
struct( ID: 82814,
        Org: "Homo sapiens",
        Seq: "ccccagcctccaacg...",
        Mod: struct(Year: 1995, Month: 1, Day: 7),
        Feats: bag(struct(Type: "source",
          FOrg: "Homo sapiens",
          Loc: var(Region: struct(From: 1, To: 8959)))))
```

[1]For simplicity, we ignore the prefix character of an accession number, i.e. we only store the number

In CPL, a database schema consists of a type and an instance of the database is a value of that type. The type of a database is typically a record type whose attributes represent entry points of the database. For example, the following type could be the schema of a genetic database that contains a set of sequence entries and a taxonomy collection:

```
struct( Entries:bag( <entry-type>),
        Taxon: bag(struct( Org: string, Parent: string,
                Descr: string)))
```

Taxonomies in biological databases are complex structures that contain information about the relationships between organisms. We consider a simplified taxonomy hierarchy: Each organism name has an associated description and a parent organism (which could be empty for the root organism classes).

2.1 Queries in CPL

The syntax of CPL used here is similar to that of SQL and OQL [6], the ODMG standard for object-oriented database languages. We adopt the select ... from ... where construct for extracting information from collections. Rather than giving the complete syntax of CPL, we will illustrate it using several examples:

Example 2.1 Select the entry ID and the last date of change of all entries that have been updated since 1996.

```
select struct(ID: e.ID, Changed: e.Mod)
  from e in Entries
where e.Mod.Year >=1996
```

The variable e is successively bound to each element of Entries. For each e, if e.Mod.Year>=1996, then the value struct(ID:e.ID,Changed:e.Mod) is constructed; all such values are then combined in a set to form the final result.

Example 2.2 Consider an UNNEST operation on the set of features: For each feature in the database, select the ID of the entry and the feature:

```
select struct (ID:id, Feat:f)
  from e as [ID id, Feats fs] in Entries, f in fs
```

Here, variable e iterates over all entries and variable f iterates over the entry's features. We use the construct e as [ID id, Feats fs] in Entries to bind variable e to the entry and variables id and fs to the entry's identifier and feature set, respectively. Expression as struct(ID id, Feats fs) is a *pattern expression*.

Patterns in CPL have two purposes. First, a pattern expresses a selection condition for values. Second, if the pattern is matched, variables can be bound within the pattern.

Example 2.3 For each entry, return the ID and the description obtained through the taxonomy collection.

```
select  struct(ID: id, TDescr: t.Descr)
  from  e as [ID id] in Entries, t in Taxon
where  t.Org=e.Org
```

In this example, e is again successively bound to each element of Entries. In addition, pattern [ID id] binds variable id to the identity of entry e. t is bound to elements of the taxonomy and the where-clause checks that the organism name of the entry equal to t.Org. This query performs a join over Taxon and Entries on Org.

Similar to OQL, we can automatically infer the type of an expression. For example, the type of Example 2.1 is bag(struct(ID:int, Changed:struct(Year:int, Month:int, Day:int))) and the type of Example 2.3 is bag(struct(ID:int, TDescr:string)).

3 Extending CPL with Updates : CPL+

Generally speaking, an update is a function from an instance of a given database schema (complex type) to another instance of that schema. Typically, there are many different ways to specify the same update. For example, it is perfectly possible in CPL to specify an update by expressing the new database value completely, as shown in the next example.

Example 3.1 Assume that a new sequence string "act" has been found and should be prepended to the sequence of the entry with ID 1234. First, the sequence has to be concatenated with the new sequence. Second, an offset of 3 has to be added to the location of features in entry 1234. We could update the set of all entries using the following select-expression:

```
Entries :=select struct(
    ID: e.ID, Mod: e.Mod, Org: e.Org,
    Seq: if e.ID=1234 then "act" + e.Seq else e.Seq,
    Feats: if e.ID=1234 then
        select struct( Type: f.Type, FOrg: f.FOrg,
                       Loc: case f.Loc of
                Reg(r) => var(Reg,struct( From: r.From+3,
                                          To: r.To+3)),
                Pos(p) => var(Pos,p+3)
        from f in e.Features
      else e.Features)
  from e in Entries
```

We use the CPL construct $\mathbf{case}\ e\ \mathbf{of}\ b_1(x_1) \Rightarrow e_1, \cdots, b_n(x_n) \Rightarrow e_n$ to operate on a variant value e. Details can be found in [16]. Clearly, this form of complete update is cumbersome to specify and inefficient if executed as written – we are rewriting the entire set of Entries, even though only a small part of the database is updated. We will therefore extend CPL with constructs for partially updating values.

Updates in CPL+ are top-level language primitives. Furthermore, CPL+ is compositional in that updates can contain "smaller" updates that are performed on subvalues in the database. An update is always executed on a given persistent value in the database, called the *update value*. The top-level update is always performed on the entire database. The most important constructs of CPL+ are given by the following grammar:

$$
\begin{aligned}
u ::= \ & \mathbf{update}\ pe\ \mathbf{with}\ u\ \mathbf{where}\ c \mid \\
& \mathbf{delete\ from}\ pe\ \mathbf{where}\ c \mid \\
& \mathbf{insert\ into}\ pe\ \mathbf{value}\ e \mid \\
& a := u \mid u_1\ ;\ u_2 \mid e \mid \mathbf{let}\ x := e\ \mathbf{in}\ u \mid (u)
\end{aligned}
$$

We use a to denote a record attribute, e to be a CPL expression, and c to be a CPL expression of type bool. Path expression pe describes a set of values that can be reached through a certain path from a given update value.

$\mathbf{update}\ pe\ \mathbf{with}\ u\ \mathbf{where}\ c$ selects the values identified by pe that satisfy c and performs u on them. $\mathbf{delete\ from}\ pe\ \mathbf{where}\ c$ deletes all elements from the set values in pe that satisfy c. The **where**-clause is optional. Update expression $\mathbf{insert\ into}\ pe\ \mathbf{value}\ e$ inserts value e into each of the set values identified by pe.

Update $a := u$ modifies attribute a of the record value by applying update u to it. An update value can be completely replaced by the new value specified through e. $u_1\ ;\ u_2$ denotes a sequence of updates. Update expression $\mathbf{let}\ x := e\ \mathbf{in}\ u$ binds the result of CPL query e to variable x and performs update u. Finally, (u) is used to form syntactic groups of updates.

Path expression pe is used to traverse paths within the database. The general syntax of path expressions is as follows:

$$pe ::= \mathbf{this} \mid a \mid pe.a \mid pe* \mid pe->a \mid pe : b \mid pe\ p$$

Expression **this** denotes the current update value itself and is the root of any path expression. A sequence of (nested) record attributes is formed using construct $pe.a$. We use single path expression a as a short-hand for $\mathbf{this}.a$. $pe*$ traverses the elements of the sets that are represented by pe. Expression $pe->a$ is a short-hand for $pe*.a$.

Expression $pe : b$ is used to traverse into a variant branch. If pe represents a set of variant values, then $pe : b$ identifies the set of branch values of those variant values that have branch b. Finally, $pe\ p$ restricts the values of pe to those that match pattern p. If a value identified by pe matches the pattern, then the variables in p are bound to the respective subvalues. We will describe patterns shortly.

Example 3.2 Set the organism name of the entry with identifier 1234 to "Homo sapiens":

```
update Entries with Org:="Homo sapiens" where ID=1234
```

Path expression Entries identifies the set of entries. Condition ID=1234 specifies a condition under which record update Organism:="Homo sapiens" is performed on the entry. Note that for each entry in Entries, the attribute names are automatically bound to the respective attribute value. Therefore, we can use ID in CPL expression ID=1234 to identify the identity of the current entry.

Each path expression identifies values of a certain type and these values can be updated using primitives **insert**, **delete**, or **update**. For example, Entries->ID represents the identifiers of all entries. To increase each identifier by 1, one could write update Entries->ID with this+1. Keyword this denotes the current update value at the end of the path. Similarly, Entries->Features represents all feature sets of all entries, this is a set of sets. Therefore, to be exact, an update of the form update Entries->Features with Type:="gene" would not be correct, since record update Type:="gene" is not a valid update for a set of features. The same update based on path expression Entries->Features*, which represents all features, would succeed.

Therefore, we adopt the following policty: The update processor automatically dereferences the last item of a path expression if the path expression identifies a set of set values, as illustrated in the previous example.

Patterns are used in path expressions to select only certain values and to bind additional variables. For example, Example 3.2 can be written as

```
update Entries[ID=1234] with Org:="Homo sapiens"
```

Here, construct [ID=1234] denotes a *pattern*. Patterns in CPL+ have the following syntax:

$$p ::= \_ \mid x \mid \ = e \mid [a_1\ p_1, \cdots, a_n\ p_n] \mid \mathsf{var}(b\ p)$$

Pattern _ is matched by any value. The variable pattern x binds variable x to the value. Pattern $= e$ only matches those values that are equal to e. Record pattern $[a_1\ p_1, \cdots, a_n\ p_n]$ matches record values whose attributes $a_1, \cdots, a_n$ match patterns $p_1, \cdots, p_n$; the record value can have other attributes in addition to those in the pattern, i.e. it matches records *partially*. Variables bound in an attribute pattern $a_i\ p_i$ can be used in subsequent attribute patterns $a_j\ p_j$ with $j > i$. For instance, the pattern $[a\ v, b = v]$ matches a record value whose attributes a and b have the same value. Finally, variant pattern $\mathsf{var}(b\ p)$ matches variant values that have branch b and whose branch value matches pattern p.

Intuitively, to select the Organism attribute of the entry with ID=1234, we would write path expression Entries[ID=1234]->Org. However, pattern [ID=1234] is not applicable to the set of all entries. Therefore, we consider path expressions $pe\ p -> a$ as a short-hand for $pe * p.a$.

Example 3.3 The previous, inefficient update on the feature range in Example 3.1 can be rewritten as the following, more intuitive CPL+ expression:

```
update Entries[ID=1234]->Seq with "act"+this ;
update Entries[ID=1234]->Feats->Loc:Reg
    with ( From:=From+3 ; To:=To+3 ) ) ;
update Entries[ID=1234]->Feats->Loc:Pos with this+3
```

The first update replaces the sequence by a concatenation of the new sequence ("act") and the old sequence. this denotes the current update value. The second and third update change the position of range and point features, respectively. Path expression Entries[ID=1234]->Seq identifies the sequence of the entry with ID=1234.

Example 3.4 Because of a recent change in naming conventions, name "Drosophilia" (Fruit fly) has to be changed to "Drosophilidae". This change affects the organism attributes of entries, of features and taxonomies.

```
update Entries->Org ="Drosophilia" with "Drosophilidae" ;
update Entries->Feats->FOrg ="Drosophilia"
    with "Drosophilidae" ;
update Taxon->Org ="Drosophilia" with "Drosophilidae" ;
update Taxon->Parent ="Drosophilia" with "Drosophilidae"
```

Example 3.5 It has been discovered that certain features in the database with description "Phaeophyceae" (brown algae) should actually carry description "Xanthophyceae" (yellow-green algae). Moreover, it has been discovered that only entries updated in 1999 are affected. In order to solve the problem, we change the organism string of all affected features.

```
update Entries[Mod [Year =1999]]->Feats
        ->FOrg="Xanthophyceae" with "Phaeophyceae"
```

It is important to note that the value of new variables in pattern p in $\mathbf{update}\ pe\ p\ \mathbf{with}\ u$ is not affected by the updates in u. We denote this as *old-value semantics*. For example, consider the following update, which can be applied to any feature record:

```
update Loc:Region r with ( From:=r.From+20 ;
                           To:=r.From+100)
```

If the feature represents a range on the sequence, then the first update From:=r.From+20 increases the start position by 20; the second update To:=r.From+100 changes the end position to be the start position plus 100. One might assume

that the first update changes the value of r.From and the second update would read the increased value. However, the value of r will remain unchanged under old-value semantics and the length of the new range is 80, and not 100. In order to express the new end position in terms of the updated start position, one has to rebind variable r after the first update:

```
update Loc:Region r with From:=r.From+20 ;
update Loc:Region r with To:=r.From+100
```

A naive approach for implementing old-value semantics is to copy the assigned value of a variable into a data container at the time of the variable binding. The value, however, can be complex and contain information that is not needed in the update process (e.g. r.To in the example above). Therefore, copying the entire content of the variable value can be inefficient. In [13], we present an efficient execution model for CPL+ updates with old-value semantics.

An update expression can only be applied to values of the correct type. For example, an update of the form Entries:=... can only be applied to record values that have a Entries attribute. Example 3.2 is only correct for record values that have an attribute Entries which is a set of records with attributes Org (of type **string**) and ID (of type **int**). An overview over the type system for CPL+ and the underlying typing rules is given in [11].

3.1 The Complex Updates Calculus $\mathcal{CUCA}$

The CPL+ update primitives provide a concise and intuitive way of specifying updates in complex value databases. In order to rewrite and optimize update expressions, we introduce a simpler and more manipulable algebraic form called $\mathcal{CUCA}$ (Complex Update CAlculus).

For every type construct, we introduce different update constructs that allow the manipulation of corresponding values. For example, collections can be manipulated by removing, modifying, or inserting elements. Record values can be modified by changing their attribute values, etc. The following grammar shows the syntax of the calculus:

$$u ::= ID_u \mid a := u \mid \langle b : u \rangle \mid \mathbf{if}\ c\ \mathbf{then}\ u_1\ \mathbf{else}\ u_2 \mid u_1\ ;\ u_2 \mid e \mid p \Rightarrow u \mid \mathbf{let}\ x := e\ \mathbf{in}\ u \mid \{\ u\ \} \mid \mathbf{delete} \mid \mathbf{insert}\ e$$

Here, e denotes a conventional CPL expression, and c is a CPL-expression of type **bool**. The construct p denotes a CPL+ *pattern matching* expression. Update expressions $a := u$, $u_1\ ;\ u_2$, e, $\mathbf{let}\ x := e\ \mathbf{in}\ u$ have the same meaning as in CPL+.

The *identity update* ID_u represents the identity function. The *variant update* $\langle b : u \rangle$ tests if the branch for the given variant value is b; if it is, then the update u is performed on the branch value. Expression $\mathbf{if}\ c\ \mathbf{then}\ u_1\ \mathbf{else}\ u_2$ evaluates condition c and, depending on the result, performs either u_1 or u_2. In the rest of the paper, we use $\mathbf{if}\ c\ \mathbf{then}\ u$ as a short-hand for $\mathbf{if}\ c\ \mathbf{then}\ u\ \mathbf{else}\ ID_u$. Update expression $p \Rightarrow u$ matches pattern p with the current update value. Patterns have the same syntax and meaning as in CPL+. If the pattern matches the current update value, then the variables in p are bound and u is performed on the update value. As we will illustrate shortly, CPL+ patterns can often be replaced by single variable bindings. In the rest of the paper, we use $\backslash x \Rightarrow u$ to denote that variable x is bound to the current update value and u is performed under this binding.

The *collection update* $\{u\}$ updates each element of a collection by u. An element of a collection can be deleted using **delete**. **delete** can only occur within a collection update $\{u\}$, such as in $\{\backslash x \Rightarrow \mathbf{if}\ x.a > 12\ \mathbf{then\ delete\ else}\ a := 23\}$. Expression $\mathbf{insert}\ e$ updates a set by inserting the elements of set e.

The conversion of CPL+ updates into $\mathcal{CUCA}$ is straightforward and omitted in this paper. Most importantly, path expressions and patterns can be eliminated.

Example 3.6 We can transform Examples 3.4 and 3.5 into the following update:

```
Entries :={ Org:= "Drosophilia" => "Drosophilidae";}
Entries :={ Feats:= {  FOrg := "Drosophilia"
                       => "Drosophilidae" }};
Taxon :={ Org:= "Drosophilia" => "Drosophilidae" };
Taxon :={ Parent := "Drosophilia" => "Drosophilidae" }
```

and

```
Entries:={ \e =>Feats:= { FOrg:= "Xanthophyceae"
  =>if e.Mod.Year=1999 then "Phaeophyceae" else ID } }
```

Note that pattern [Mod [Year=1999]] in Example 3.5 is transformed into a variable binding \ e =>... and a pattern condition e.Mod.Year=1999.

Example 3.7 Consider both updates in Example 3.3. The corresponding $\mathcal{CUCA}$ updates have the following form:

```
Entries:= { [ID=1234] => Seq:= \ s =>"act"+s } ;
Entries:= { [ID=1234] => Feats:={ Loc:=
        <Reg : \ r => (From:=r.From+3; To:=r.To+3) >}} ;
Entries:= { [ID=1234] => Feats:={ Loc:=
        <Pos : \ p => p+3 >}}
```

4 Rewrite Rules and Optimization

The optimization of updates in databases is a complex and intriguing problem. As with the optimization of queries, the optimization of updates can be achieved by several means: rewrite rules, optimizations based on statistics,

and the clever use of caching, clustering and indexing. Here we concentrate on the use of rewrite rules to take an update expression and transform it into a faster, yet equivalent, expression. The correctness of these update rules has been proven in [11], and a storage and execution model is presented in [13] along with proofs that the rules are improving. This section summarizes this work by presenting the rewrite rules and illustrating their effectiveness on some examples.

The measure of update cost used in [11] and [13] is based on the notion that a complex value database forms a *value tree*. Leaf nodes are primitive values such as string constants or boolean values. Record values have sub-nodes, each of which represents an attribute value; edges are labelled with the record label. Variant values have a sub-node representing the value of the branch; the edge is labeled with the variant label. Set nodes have a set of subnodes, each of which represents exactly one element of the set; edges are unlabeled.

In an efficient implementation, neighboring nodes will be stored together on single pages or within clusters. Often values will be fixed-length, making the storage structures much simpler and more efficient. Other cost reducing techniques such as caching, indexing and deferred updating can also be applied. However, for now we ignore these other types of optimizations and use an obvious and naive execution model for CPL+. In particular: The construct $\mathbf{if}\ c\ \mathbf{then}\ u_1\ \mathbf{else}\ u_2$ evaluates c and performs either u_1 or u_2. The set update $\{u\}$ iterates over the elements (i.e. sub-nodes) of the set node and performs the update u on each of them. The complete update e replaces the old tree of the updated value by a new tree which is the result of evaluating e.

Updating a database therefore 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, additional objects (i.e. nodes in the tree) of the database may have to be retrieved between the updates. Although the actual cost of an update differs from implementation to implementation, we can assume that the cost is proportional to the number of objects "touched" within the execution of an update – i.e. the collection of objects read, written, created or deleted, denoted as the *working space* of an update. The rewrite rules presented therefore attempt to improve the performance of the update by reducing its working space. Proofs that these rules are correct and reduce the working space can be found in [11]. We present two classes of optimizations: optimization by rewriting and deltafication.

4.1 Optimizing Update Expressions by Rewriting

The semantics of CPL+ implies a variety of rewriting rules. Many of them do not reduce the cost of the update directly, but enable a series of transformations that 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.

4.1.1 Update Fusion

Sequences of updates on the same value can be merged. As a general rule, it is desirable to move the sequence operator to the leaves of the syntax tree so that updates to the same value are clustered together and further cost-reducing simplifications can be applied.

$$
\begin{array}{rcl}
\langle b : u_1 \rangle\ ;\ \langle b : u_2 \rangle & \Rightarrow & \langle b : u_1\ ;\ u_2 \rangle \\
\langle b_1 : u_1 \rangle\ ;\ \langle b_2 : u_2 \rangle & \Rightarrow & \langle b_2 : u_2 \rangle\ ;\ \langle b_1 : u_1 \rangle \\
(a := u_1)\ ;\ (a := u_2) & \Rightarrow & a := (u_1\ ;\ u_2) \\
(a_1 := u_1)\ ;\ (a_2 := u_2) & \Rightarrow & (a_2 := u_2)\ ;\ (a_1 := u_1) \\
\{u_1\}\ ;\ \{u_2\} & \Rightarrow & \{u_1\ ;\ u_2\} \\
(\mathbf{let}\ x := e\ \mathbf{in}\ u_1)\ ;\ u_2 & \Rightarrow & \mathbf{let}\ x := e\ \mathbf{in}\ (u_1\ ;\ u_2) \\
(p \Rightarrow u_1)\ ;\ u_2 & \Rightarrow & p \Rightarrow (u_1\ ;\ u_2)
\end{array}
$$

Figure 2. Update Fusion

The second and the fourth rule are only correct if $b_1 \neq b_2$ and $a_1 \neq a_2$ hold, respectively. They are used to cluster updates on the same label together so that the first and third rule can be applied. The last rule in Figure 2 is only applicable if the pattern is not restricting, i.e. every type-correct update value matches the pattern. This is the case if the pattern does not contain a CPL expression e or $\mathsf{var}(b : p)$.

The rule $\{u_1\}; \{u_2\} \Rightarrow \{u_1; u_2\}$ is called *vertical loop fusion* and is a powerful optimization rule, as illustrated in the following two examples:

Example 4.1 Consider the sequence of updates in Example 3.6:

```
Entries :={ Org := "Drosophilia" => "Drosophilidae" } ;
Entries :={ Feats := { Descr:= "Drosophilia"
        => "Drosophilidae" }} ;
Taxon :={ Org :="Drosophilia" => "Drosophilidae" } ;
Taxon :={ Parent:= "Drosophilia" => "Drosophilidae" } ;
Entries :={ \e => Feats:= { Descr:= "Xanthophyceae" =>
        if e.Mod.Year=1999 then "Phaeophyceae" }}
```

In this update, the processor has to iterate three times over the entries and two times over the taxonomy collection. Furthermore, for each entry, all features have to be considered. Using the rules for update fusion, we will obtain the following, more efficient update:

```
Entries :={\ e => Org:= "Drosophilia" => "Drosophilidae" ;
      Feats:= { Descr:= ( "Drosophilia" => "Drosophilidae" ;
                          "Xanthophyceae" =>
      if e.Mod.Year =1999 then "Phaeophyceae" )}};
 Taxon :={ Org:="Drosophilia" => "Drosophilidae" ;
           Parent:="Drosophilia" => "Drosophilidae" }
```

One can easily observe that we only need one iteration over entries, each of their features, and the taxonomy collection. This can lead to substantially lower update cost.

Example 4.2 Consider the updates in Example 3.3 and 3.7. Executing the update will lead to three iterations over Entries. Using the update fusion rules, we can rewrite the update sequence to the following, more efficient update:

```
Entries:= {[ID=1234] =>( Seq:= \ s =>"act"+s ;
                         Feats:={Loc:=(
    <Reg: \ r => (From:=r.From+3; To:=r.To+3)>;
    <Pos: \ p => p+3 >)})}
```

In order to do the transformation we use the fact that condition x.ID=1234 is equivalent to condition y.ID=1234 in the second update. Furthermore, the value x.ID is not changed in the first update. Therefore, both patterns operate indeed on the same value and, therefore, can be merged. This optimization is done by reasoning over the equivalence of predicates and patterns. We omit the details of this optimization.

In practice, the performance gain is influenced by the storage model and indices used to access data. In our example, if an index on ID is used for a fast lookup of the corresponding entry then the update fusion optimization only leads to a small cost improvement. In general, the existence of indices cannot be guaranteed.

4.1.2 Filter Promotion

Similar to rewriting in the relational algebra, it is desirable to perform selections as early as possible. In this way, unnecessary evaluations can be avoided. In CPL+, filters (i.e. expressions $\mathbf{if}\ c\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$) are moved to the outside of complex updates, as shown in Figure 3.

Function $f_B(p)$ denotes the set of variables that are newly bound in pattern p. Function $f_V(e)$ denotes the set of free variables in CPL expression e.

To illustrate these rules, consider the second update in Example 3.6, which changes the organism description of features from "Xanthophyceae" to "Phaeophyceae" if the corresponding entry has been updated in 1999:

```
Entries := { \ e => Feats:= { Descr:= "Xanthophyceae" =>
    if e.Mod.Year =1999 then "Phaeophyceae"
```

Clearly, it would be more efficient to check e.Mod *before* the iteration over all features. Using the rules in Figure 3, we can indeed rewrite the update to the following expression:

```
Entries := { \ e => if e.Mod.Year=1999 then Feats:=
   { Descr:= "Xanthophyceae" =>"Phaeophyceae" }}
```

As a result, only the features of entries that were modified in 1999 are considered.

The optimizations rules marked with $*$ will under certain conditions *increase* the cost of the update. In the original expressions, it is possible that condition c is not evaluated at all: For the four marked rules, this is case if 1) u_1 deletes the update value from a set using $\mathbf{delete}$, 2) the variant does not have branch b, 3) if the set is empty or 4) if the pattern p is not matched, respectively. The rewritten expressions, however, lead to an evaluation of c in every case. In the worst case, such an evaluation can have high cost.

On the other hand, the rule for $\{\mathbf{if}\ c\ \mathbf{then}\ u_1\ \mathbf{else}\ u_2\}$ can lead to significant cost improvements if the set has a lot of elements, since c is only evaluated once instead of multiple times. While in general we assume that the average cost reduction achieved outweighs the average cost increase, it is clear that these rules are best used with instance-level statistics to guide their application. This is a topic for future research.

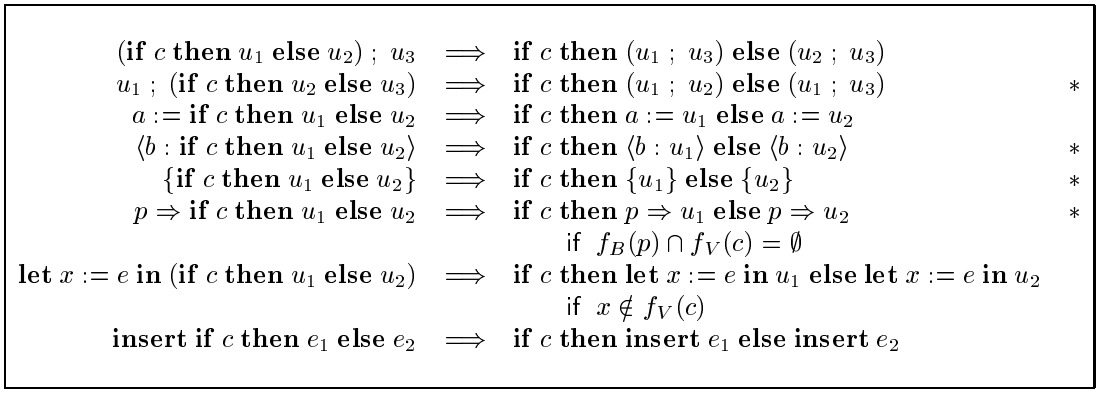
4.1.3 Other Optimizations by Rewriting

Beside update fusion and filter promotion, a variety of other optimizations are available. Patterns and in particular single variable binding can be often be simplified or eliminated. Furthermore, update constructs containing primitive updates such as $\mathbf{insert}\ e$, $\mathbf{delete}$, complete update e, or ID_u can often be simplified. Due to space restrictions, we omit the details. A more exhaustive description of rewrite rules can be found in [11].

4.2 Deltafication

Update fusion and filter promotion reduce the number of objects read during the update process. In this section, we will describe a mechanism to transform inefficient "complete" updates into more efficient updates. The basic principle is that deltafication analyses a given CPL expression which replaces a previous value in the database and identifies the actual *changes* that have to be made to the database. We distinguish two types of deltafications: *abstract deltafication* and *concrete deltafication*.

Abstract Deltafication works solely on complete update expressions specified in CPL. A complete update evaluates



$$
\begin{array}{rcll}
(\textbf{if } c \textbf{ then } u_1 \textbf{ else } u_2)\ ;\ u_3 & \Longrightarrow & \textbf{if } c \textbf{ then } (u_1\ ;\ u_3) \textbf{ else } (u_2\ ;\ u_3) & \\
u_1\ ;\ (\textbf{if } c \textbf{ then } u_2 \textbf{ else } u_3) & \Longrightarrow & \textbf{if } c \textbf{ then } (u_1\ ;\ u_2) \textbf{ else } (u_1\ ;\ u_3) & * \\
a := \textbf{if } c \textbf{ then } u_1 \textbf{ else } u_2 & \Longrightarrow & \textbf{if } c \textbf{ then } a := u_1 \textbf{ else } a := u_2 & \\
\langle b : \textbf{if } c \textbf{ then } u_1 \textbf{ else } u_2 \rangle & \Longrightarrow & \textbf{if } c \textbf{ then } \langle b : u_1 \rangle \textbf{ else } \langle b : u_2 \rangle & * \\
\{\textbf{if } c \textbf{ then } u_1 \textbf{ else } u_2\} & \Longrightarrow & \textbf{if } c \textbf{ then } \{u_1\} \textbf{ else } \{u_2\} & * \\
p \Rightarrow \textbf{if } c \textbf{ then } u_1 \textbf{ else } u_2 & \Longrightarrow & \textbf{if } c \textbf{ then } p \Rightarrow u_1 \textbf{ else } p \Rightarrow u_2 & * \\
 & & \quad \text{if } f_B(p) \cap f_V(c) = \emptyset & \\
\textbf{let } x := e \textbf{ in } (\textbf{if } c \textbf{ then } u_1 \textbf{ else } u_2) & \Longrightarrow & \textbf{if } c \textbf{ then let } x := e \textbf{ in } u_1 \textbf{ else let } x := e \textbf{ in } u_2 & \\
 & & \quad \text{if } x \notin f_V(c) & \\
\textbf{insert if } c \textbf{ then } e_1 \textbf{ else } e_2 & \Longrightarrow & \textbf{if } c \textbf{ then insert } e_1 \textbf{ else insert } e_2 &
\end{array}
$$

Figure 3. Filter Promotion

the CPL query and replaces the current update value by the result of the query. Most likely, the query is based on the value itself and the result of the query does not differ much from the original update value. Abstract deltafication identifies this change and generates an appropriate CPL+ expression.

Consider our original, inefficient Example 3.1. Using the deltafication rules described in [11], we can automatically transform this complete update into a more efficient $\mathcal{CUCA}$-expression:

```
Entries:={ \ e => if e.ID=1234 then
  Seq:= \ s => "act"+s ;
  Feats:={ Loc:= <Reg:(From:= \ f => f+3 ; To:= \ t => t+3)>;
                 <Pos:\ p =>p+3 >}}
```

This update is equivalent to the CPL+ expression shown in Example 4.2.

Abstract deltafication is based on a set of rules that recursively transform query expressions into corresponding delta updates [11]. These rules are based on the internal representation of CPL expressions, called $\mathcal{NRC}^+$ (Nested Relational Calculus) [16].

Concrete Deltafication identifies the changes between two specific values of the same type and generates a CPL+ expression. In databases such as Genbank and SwissProt, updates on entries are often specified by providing the complete new entry. Typically, the changes made to each updated entry are rather small. Therefore, updating the database by replacing the entire entry can be expensive.

For example, consider a given Genbank entry with ID=82814, where the modification date has been changed to June 11, 1999, the organism name has been changed to "Homo sapiens neanderthalensis", the end position of a source feature with range 1...8959 has been changed to position 8961, and a new gene feature has been inserted. Giving the old value of the entry and the updated entry value, the concrete deltafication would generated the following update:

```
Entries:= { [ID=82814] =>
   Mod:= (Year:=1999; Month:=2; Day:=10);
   Org:= "Homo sapiens neanderthalensis";
   Feats:= { [Type="source", Loc <Reg [From=1,To=8959]>]
     =>Loc:=<Reg: To:=8961>};
      insert set(struct(Type: "gene",
            Loc: var(Reg, struct(From: 11, To: 187)))) }
```

The problem of detecting the changes between two string has been investigated extensively [14, 15]. It has important applications in various scientific areas, such as in sequence matching for genetic databases. More recently, the problem of identifying changes in arbitrary nested structures, such as ordered and unordered trees, has been analyzed [17, 18, 7]. Slight modifications of these algorithms are being used to generate CPL+ expressions during concrete deltafication.

5 Conclusions and Future Work

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 on complex values. In particular, it provides set-oriented updates (analogous to those in SQL-92) as well as mechanisms for specifying changes "deep inside" the database, i.e. paths and patterns. CPL+ was illustrated using typical forms of updates in biomedical databases, a domain for which it is ideally suited due to the deeply nested, complex types.

CPL+ also admits a variety of optimizations. To specify optimizations, we introduced a more manipulable update calculus, $\mathcal{CUCA}$. Using $\mathcal{CUCA}$, various rewriting

rules, such as update fusion and filter promotion, were presented. In addition, we describe a method for transforming complete updates (i.e. updates that rewrite an entire complex value) into more efficient updates that only change the necessary parts of the value. This optimization, called *deltafication*, has two different forms: *abstract* and *concrete deltafication*. Concrete deltafication is used in conjunction with structure differencing algorithms to produce minimal changes between one version of an entry and another.

An in-depth description of CPL+, including a complete set of optimization rules and proofs of correctness and cost reduction, can be found in [12, 11]. A prototype for CPL+ has been implemented which is based on an abstract execution model described in [13]. The prototype implements the optimization rules presented in this paper (including deltafication), and uses a type inference algorithm based on the typing rules given in [11]. Based on the notion of an update's *working space* – i.e. the set of objects that are read, written, deleted or inserted within an the execution of an update – we were able to prove that the cost of update does not increase during optimization, except in the special cases mentioned in this paper.

Various other interesting practical and theoretical problems remain to be investigated. Since CPL is currently being used for querying multiple, heterogeneous database systems [8], the issue of what updates across many databases mean and how to optimize them has to be addressed. Another important issue is how to detect and to resolve conflicts between updates.

Acknowledgments: We would like to thank Peter Buneman for help with the original update language, and Val Tannen for many helpful discussions.

References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-96*. Morgan Kaufmann, San Mateo, California, 1996.

[7] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 26–37, Portland, Oregon, 1997.

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

[9] N. Goodman, S. Rozen, and L. Stein. Requirements for a deductive query language in the MapBase genome-mapping database. In *Proceedings of Workshop on Programming with Logic Databases, Vancouver, BC*, October 1993.

[10] 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.

[11] H. Liefke and S.B. Davidson. An execution model for CPL+. Technical Report MS-CIS-98-29, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1998.

[12] 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, May 1998.

[13] H. Liefke and S.B. Davidson. Processing updates in complex value databases. In *Proceedings of IRMA International Conference*, pages 333–342, Hershey, PA, May 1999.

[14] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[15] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, January 74.

[16] 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.

[17] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.

[18] K. Zhang, J. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs. *International Journal of Foundations of Computer Science*, 1995.